NTNU

Norwegian University of
Science and Technology

Anonymous Authentication Using Secure Multi-Party Computations

Maqsood Ahmad

Master in Security and Mobile Computing
Submission date: June 2011
Supervisor: Stig Frode Mjølsnes, ITEM
Co-supervisor: Tord Reistad, ITEM

Norwegian University of Science and Technology
Department of Telematics

Anonymous Authentication Using Secure Multi-Party Computations

Maqsood Ahmad

Academic Advisors:

Prof. Stig F. Mjølsnes
Norwegian University of Science and Technology, Norway

&

Prof. Gerald Q. Maguire Jr.
Royal Institute of Technology, Sweden

Supervisor:

Tord I. Reistad

June 30, 2011

Problem Description

Secure multi-party computation (MPC) is a method whereby multiple players (computers) can interact to compute a common function based on secret input that each player has. Computing this common function is done without revealing each player's secret. MPC was used last year to implement the RSA algorithm without any of the players ever knowing the primes p or q and distributed voting. This year the challenge is to design a model for a distributed anonymous authentication system. For example a user authenticates himself in such a way that no single server he connects to knows who was authenticated only that it was a valid authentication. The user is then provided with access to various services by the system.

The project will give the student an understanding of multi-party computation and the possibility to implement the algorithm or system using VIFF (www.viff.dk)

Assignment given: January, 2011

Professors: Stig Frode Mjølsnes and Gerald Q. Maguire

Supervisor: Reistad Tord Ingolf

Abstract

Typical authentication systems provide a method to allow registered users access to protected resources after the user successfully authenticates. A user successfully authenticates by proving his or her valid identity if he or she is a registered user. During a typical authentication process, the authentication server can directly or indirectly learn the actual identity of the user who authenticates. However, the user might not want any one to know the actual identity of the user, while still able to authenticate. This problem of user's anonymous authentication is the focus of this thesis project. This thesis project provides a solution for user's anonymous authentication using Secure Multi-party Computation (SMPC). In SMPC, the user information is distributed among the authentication servers, using a secret sharing scheme, in such a way that none of the authentication server individually possesses all the information of a user. However, these authentication servers can validate the user using some SMPC arithmetic operations. This thesis project provides a model for anonymous authentication and couples this anonymous authentication system with the Open Authentication Protocol (OAuth) to allow the user access to protected resources on the server. The model is explained using UML collaborations and SDL state transition diagrams. A analysis of the model is provided to ensure the security of the proposed system. A skeleton of the proposed model is provided which needs to be completed with appropriate code to realize the functionalities. This thesis project also provides an implementation of a simplified prototype which represents the core of the proposed model for anonymous authentication.

Acknowledgment

Completing a NordSecMob masters thesis in a consistent and efficient manner becomes quite a challenging task when working under the supervision of professors from different universities, but I appreciate and acknowledge my professors and supervisor both at NTNU and KTH for their valuable feedback and support during my master thesis.

I would like to thank Prof. Gerald Q. Maguire and Prof. Stig Frode. Mjølsnes (my academic advisors at KTH and NTNU respectively) for their timely, specific, and detailed feedback and comments on my report. Their feedback proved very fruitful for me in completing my masters thesis.

I would also like to acknowledge the support and directions I received from my supervisor Tord I. Reistad throughout my masters thesis. His calm nature and easy approach to difficult problems was a great inspiration for me during my thesis.

At last, I would like to specially thank my parents, family members, and my friends who have always been there for me throughout my life.

Contents

List of Figures

Listings

List of Acronyms

AES Advanced Encryption Standard
API Application Programming Interface
AnonAuth Anonymous Authenticator

AnonAuthenticate Anonymously Authenticate
AnonReg Anonymous Registration
AnonRegisters Anonymously Registers
AuthAck Authentication Acknowledgment
AuthNAck Authentication Negative Acknowledgment
GWServer Gateway Server
GWServerSM Gateway Server State Machine
HTTP Hypertext Transfer Protocol
ID Identifier
MPC Multi-party Computation
OAuth Open Authentication Protocol
OT_Key One time Key
U_Key User Key
U_ID User ID
PRSS Pseudo Random Sharing Scheme
RFC Request for Comments
RSA Rivest, Shamir, and Adleman
RegAck Registration Acknowledgment
RegNAck Registration Negative Acknowledgment
RegRequest Registration Request
RegUser Registers User
RegisterUser Register User
RegistrarSM Registrar State Machine
ReqAuthenticate Request for Authentication
ReqAuthorize Request for Authorization
ReqCred Request for Credentials
SDL Specification and Description Language
SIMAP Secure Information Management and Processing
SMPC Secure Multi-Party Computation
SSL Secure Socket Layer
TempCred Temporary Credentials
TokenCred Token Credentials
UML Unified Modeling Language

URI Uniform Resource Identifier
UnionServer Union Server
CompServer Company Server
VIFF Virtual Ideal Functionality Framework

Chapter 1

Introduction

A major portion of the population make use of a variety of services provided by different service providers. Depending upon the nature of the service, the goal of the service provider, and requirements of the service; one group of users can be distinguished from another and user can be put into different classes. Users belonging to a certain class are provided with a specified level of access. The service provider maintains a database of users. The users normally provide some information that is stored in the database when they register with the service provider. The service provided by the service provided can later be accessed by registered users after authenticating themselves to a server. The server authenticates a user based on the credentials provided by the user. These credentials can be in the form of a Username and Password, some sort of user specific secret key, etc. The authenticated user is then allowed to access the service hosted by the server.

1.1 Context of the Problem

The credentials used in a typical authentication process is related to the user's identity and either directly or indirectly identifies the user. So the authentication process enables the authenticating server to know about the identity of the user along with allowing it to decide whether the user is a valid user or not. Most of these services require that the user's ID should not only be known to the server but to other users as well. An example of

such kind of services can be an email service provided by a company. Here the company's webmail server should know the identity of the user *before* registering this user with the service and the other users in the company should also know the user's identity in order to communicate with him. The same company might provide some other services which require the user's ID to be known to the server, but this identity might not be of any concern to the other users, e.g an Internet service where different users are provided with different bandwidth and quality of service.

In contrast to the above mentioned services, it might not be important to know the identity of the user as long as the service provider can be assured that this specific potential user is a valid user. A valid user should be able to make use of the provided service without anybody knowing the user's actual identity. An example service can be a 'Whistle Blower' service. Any problem in the company can be reported by a valid company employee but the employee is able to conceal his or her identity. Such an anonymous service is important as the company is concerned with having problems reported, rather than who reports the problem. The only concern is that the person who reports the problem should be an employee of the company.

1.2 Motivation for the solution

Implementation of the above service and other like it needs to consider two important issues. The first issue is the anonymity of the user and the second one is the authenticity of the user. A valid user must be able to report a problem inside the company anonymously. Anonymity may be essential for the safety, security, and personal integrity of the user. The reported problem might be related to one of the company's managers and this manager might harm the person reporting the problem if the reporter's identity were to be known. Anonymity ensures that the report can be made, but the source of the report can not be known. Authenticity of the user is also very important and only a valid company user should be able to use the 'Whistle Blower' service; otherwise an intruder from outside could report an imaginary problem or fabricate false accusations and create panic in the company wasting scarce company resources and potentially impairing productivity of the employees.

1.3 Multi-Party Computation as a potential solution

The problem described in the previous section can be solved using a system that can authenticate users anonymously, i.e., splitting authentication away from identification. A manual solution for this is to print anonymous IDs and a corresponding password on pieces of paper and let everybody in company take one or more of them. The system can then utilize these IDs and passwords to authenticate users without knowing their actual identities. This solution might be applicable for a small company with very few employees, but it becomes more and more unmanageable as the number of employees in the company increases. Also, this solution does not provide a back door for tracking the user in case of a serious offense made by the user.

Another solution is to have a trusted third party between the employees and company. The third party is provided with access to the company's user database and it is responsible for authenticating the user to use a service provided by the company. Both the parties, i.e. employees and the company, must trust the third party. The employee trusts the third party that it will not reveal his or her actual identity and the company trusts the third party that it will not authenticate invalid users. This solution has its own problems. The company has to acquire the services of a third party and bear the expenses. Involvement of a third party necessarily implies that the company must expose a portion of its user's information to someone who does not belong to the company. The third party can also be influenced or compromised by some malicious person or persons who could harm the company's cause.

So a trusted third party apparently solves the problem, but it raises a lot of other issues. However, a trusted third party can be replaced with Secure Multi-Party Computation (SMPC) [1]. Potentially Secure Multi-Party Computation can resolve the issues and concerns raised when using a trusted third party. This thesis focuses on how Secure Multi-Party Computation can be used to anonymously authenticate a user, thus enabling a user to access one or more services with the help of the Open Authentication Protocol (OAuth) [2].

1.4 Goal of the project

The goal of this project is to utilize Secure Multi-party Computation to enable anonymous user authentication. To demonstrate that this is feasible required designing a model, performing a security analysis of the model, and implementing a prototype based on Secure Multi-party Computation which can be used to build a system where users can authenticate themselves anonymously. After successful authentication a user should be allowed to access specific services using the Open Authentication Protocol. The resulting model, design and prototype should be evaluated.

1.5 Contributions

This thesis project describes Secure Multi-party Computations (SMPC), Virtual Ideal Functionality Framework (VIFF), and the Open Authentication Protocol (OAuth) in order to understand how these concepts can be used to design a model for a user's anonymous authentication. The main contributions of this thesis project are:

- Designing a system for anonymous authentication using SMPC,
- Elaborating the proposed system using UML collaboration models and SDL state transition diagrams,
- Analyzing the proposed system to ensure security,
- Providing a skeleton of the proposed system which needs to be completed with appropriate code to realize the functionalities in order to implement the complete system, and
- Implementing a VIFF based simplified prototype for anonymous authentication.

1.6 Outline of the report

The thesis first lays some background about the problem and the techniques that will be later user for the actual solution of the problem. The major building blocks of the proposed solution are explained in Chapter 2 along with a brief review of related work. Chapter 3 provides a detailed explanation of the proposed model for the Anonymous Authentication using UML collaboration diagrams and SDL state transition diagrams. Chapter 4 focuses on a security analysis of the proposed model and discusses how well the model achieves the goals of the project. The necessary guidelines for the development of the proposed authentication system are provided in Chapter 5. Implementation of a simplified prototype is presented in Chapter 6. Chapter 7 concludes this document and suggests future work to follow up this research.

Chapter 2

Background and related work

This chapter establishes a base for this thesis by explaining the major pillars which support the the proposed model. The three major sections in this chapter explain Secure Multi-Party Computation, the Virtual Ideal Functionality Framework, and the Open Authentication Protocol. The chapter also provides a brief review of the work already done in these areas.

2.1 Secure Multi-Party Computation

Multi-party Computation (MPC) involves multiple parties who want to compute a specific agreed function based on certain inputs from each party in such a way that none of the parties know the input of any other party. The only information each party acquires from the computation is the public output of the MPC. The first MPC problem introduced was the Millionaire problem where two millionaires want to know who is richer but do not want to reveal any information about their wealth to each other [3].

2.1.1 Trusted third party and SMPC

These types of problems can be solved by involving a trusted third party by each of the parties providing their inputs to the trusted third party and

the third party reveals the public output of the commonly agreed function based on the inputs provided by the participating parties. However, MPC replaces the trusted third party thereby avoiding the problematic issues of a third party solution. The individual inputs are shared with the other parties in such a way that every party is able to compute the agreed function and learn the desired output but still remains unaware of the actual inputs of the other parties. This is made possible with the help of a concept called verifiable secret sharing [4][5].

Each of the parties splits its secret, before sharing it, into as many parts as the number of parties take part in the MPC. In this way every party gets a share of that secret, but not the complete. All the parties share their secrets in the same manner. A secret can only be reconstructed if a party has more than a specific number of shares greater than a certain threshold. However, every party is capable of performing computations such as addition, multiplication, and comparison on the secret shared values without knowing the actual secrets. These computations are used to compute the commonly agreed function.

2.1.2 Secret Management

The main feature of MPC is to hide the individual's secret values from one other, while still providing them with a method to do computations based on these secrets values. This is done using a secret management process. This process manages the splitting and sharing of the secrets as well their reconstruction.

2.1.2.1 Secret Sharing

Secrets are shared among the participants in such a manner that the individual shares do not reveal any information about the secret. Consider the case of three parties named User1, User2, and User3 taking part in a MPC as shown in Figure 2.1. X, Y, and Z are the secrets of User1, User2, and User3 respectively. Each of these parties divides its secret in three parts, e.g. User1 divides its secrets X into three parts x1, x2, and x3. User1 keeps x1 to itself and sends x2 and x3 to User2 and User3 respectively. After the exchange, every party has a share of every other party's secret which it can

use to do certain kinds of computations as part of the commonly agreed function.

There are various schemes available for secret sharing. Splitting a secret directly into multiple parts is one of the easiest schemes, but it is not a secure because an adversary an adversary can reconstruct the secret if the adversary learns some of the shares. Therefore the secrets must be divided in a such a way that the secret can only be reconstructed when a certain number of shares greater than a specific threshold are known. Each individual share also must not reveal any information about the secret.

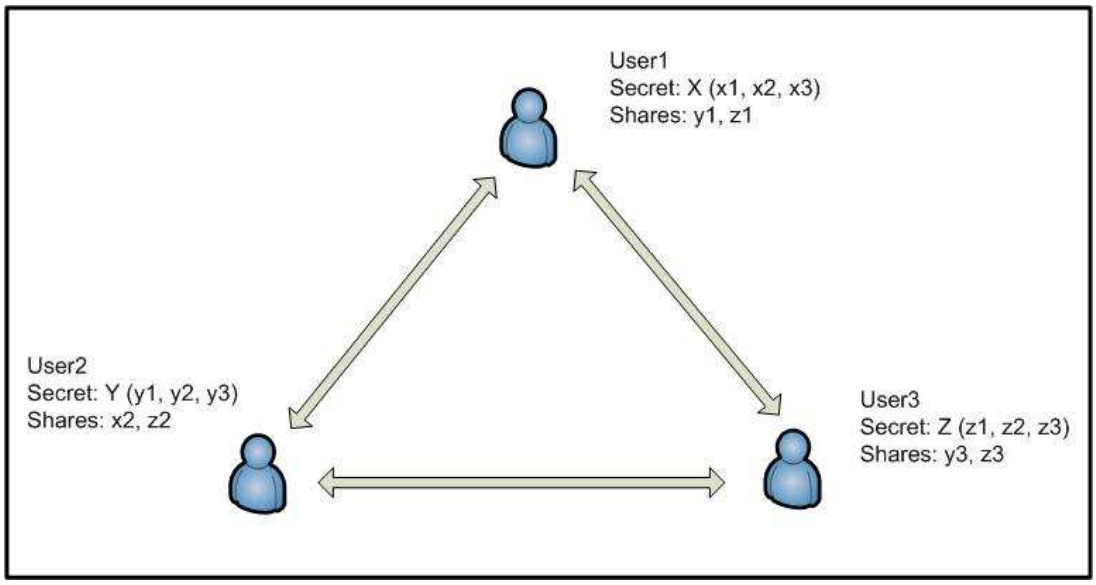

Figure 2.1: Secret splitting and Sharing

Some restrictions on the secret sharing scheme can help in fulfilling the aforementioned conditions as stated in [6]. One restriction is that the size of each share must be equal to the original secret and no information can be obtained until more than a certain number of shares are known. The sharing scheme must also use random bits in the formation of shares.

The number of shares required to reconstruct the secret is termed the threshold t in most of the literature. This implies that we can have less than t adversaries in a MPC *without* affecting the final result. In an n-party computation, t varies from n to $n/2$ to $n/3$ depending upon the sharing scheme used and the nature of adversaries [7][8][9]. The following two subsections describe two sharing schemes: Additive Sharing Scheme and

Threshold Sharing Scheme [10][11].

2.1.2.2 Additive Sharing Scheme

The additive sharing scheme is the simplest sharing scheme. It is an absolutely secure sharing scheme if done with care. It requires a knowledge of all the shares to reconstruct the secret and therefore this scheme is also called the perfect secret sharing scheme. Additive sharing schemes are methods for creating shares and reconstructing the secret from the shares [12].

If there is a secret s to be shared among n parties, n-1 random numbers r_1 to r_{n-1} are selected. The random values will be used to compute the shares corresponding to the n-1 parties. A prime number p greater than all the random numbers and the secret is selected and then the secret s and the random numbers r_1 to r_{n-1} are considered to be element from within the set S mod p in order to make the selection of the values equally likely. The share for the nth party is computed using equation 2.1 [12].

$$r_n = s - \sum_{i=1}^{n-1} r_i \, mod \, p \tag{2.1}$$

The secret can be reconstructed using equation 2.2 only when all the shares are known [12].

$$s = \sum_{i=1}^{n} r_i \, mod \, p \tag{2.2}$$

A variant of the additive sharing scheme is the XOR sharing scheme which works in exactly the same manner as the additive sharing scheme. The n-1 shares are random numbers and the nth share is computed such that the XOR of all the shares is equal to the secret s, i.e., the nth share is the XOR of all the n-1 shares and the secret s itself. The secret s can be reconstructed by XORing all the n shares [13].

The reconstruction of the secret requires the knowledge of all the n shares, so an adversary having the knowledge of less than n shares will not be able

to reconstruct the secret. However, there is a problem if even one share is missing, as no one is able to reconstruct the secret. This problem can be solved by the sharing scheme described in the next section.

2.1.2.3 Threshold Sharing Scheme

Threshold sharing schemes uses a polynomial to create shares from a secret and to reconstruct the secret from these shares. Such a scheme is also be called a polynomial sharing scheme or a (t,n)-threshold scheme where n is number of parties in a n-party computation among whom the secret s is shared in such a way that any number of parties less than the threshold t cannot reconstruct the secret [12].

Shamir came up with a sharing scheme based on polynomial interpolation known now as the Shamir sharing scheme. The Shamir sharing scheme is used in a framework for MPC called Virtual Ideal Functionality Framework (VIFF). VIFF will be discussed later in section 2.2 [15]. In the Shamir sharing scheme, a secret is represented by a polynomial where all the numbers in the polynomials are from a finite field, i.e., a field with a finite number of elements and having a prime number or power of a prime number p as its degree. A modulo p secret sharing scheme ensures that the secret can be any element from the ring of integers modulo p, whereas the same cannot be stated for all the real numbers because the secrets may not be uniformly distributed across all the real numbers.

The polynomial is constructed based the variable t in the (t,n)-threshold scheme. A (t,n)-threshold scheme can be represented by a polynomial of degree t-1 [11]. The secret s is represented by a point $(0,s)$ on the y-axis where the polynomial intersects the y-axis. For example a $(2,n)$-threshold scheme is represented by line where $(0,s)$ is the point on y-axis representing the secret and a $(3,n)$-threshold scheme is represented by a quadratic equation where $(0,s)$ is a point on the y-axis, representing the secret, where the curve represented by the quadratic equation intersects the y-axis. The n shares are any other n points that satisfy the polynomial. Similarly higher threshold schemes require higher degree polynomials.

The secret can be reconstructed having the knowledge of any t shares using Lagrange's formula as shown by equation 2.3 where s is the secret, f represents the polynomial and $f(x) = s_x$ is the share corresponding to xth

player [14].

$$s = \sum_{i=1}^{t} s_i \prod_{j=1\ j!=i}^{t} -x_j/(x_i - x_j)\ mod\ p \tag{2.3}$$

2.1.3 Arithmetic on secret shared values

The commonly agreed functions used in MPCs are based on arithmetic operations such as addition, multiplication, etc. VIFF, to be discussed later, provides support for all these arithmetic operations.

2.1.3.1 Addition

Addition of secret shared values is very simple. Each party can find the sum of the secrets by adding their individual shares of the secrets. Secrets are shared using particular polynomials. The sum of two or more polynomials generate a new polynomial which has the coefficients equal to the sum of the corresponding coefficients in the polynomials to be added up. The sum of the polynomials intersect the y-axis at the same point as the sum of the secrets represented by them.

2.1.3.2 Multiplication

Multiplication of secret shared values is more complicated as compared to addition of the secret shared values. It requires multiple layers of sharing and some extra calculations to compute the product of two secret shared values [6]. The details are not discussed here because that is not relevant to the goal the thesis.

2.1.3.3 Comparison

Comparison of secret shared values is much more complicated than either addition or multiplication of secret shared values. A comparison operation

consists of multiple multiplication operations which makes it a very slow and expensive process although researcher are trying to make the comparison operation faster and cheaper.

This thesis focuses on one variant of comparison, specifically the equality of secret shared values. There are different methods to find out whether the secret shared number are equal or not. A bitwise XOR operation or computing the difference of two secret shared numbers help in determining whether these number are equal or not.

2.1.4 SMPC Work flow

We can now organize the above concepts and procedures to form a work flow of the SMPC. SMPC is performed in three steps: input sharing, computing the desired function, and revealing the output.

Input Sharing The secret are divided into shares and shared with other parties according the sharing scheme, the number of parties, and the requirements of the specific application.

Computing the desired function SMPC is based on some function which all the parties agree upon. This commonly agreed function can be based upon arithmetic operations as discussed above. The necessary computations required to compute the desired function are performed and final value of the function is computed.

Revealing the Output The output of the previous stage is revealed to all or some of the parties at this stage based on the requirements of the protocol. There can also be additional actions that are taken based on the previous stage's output in order to meet the requirements of a particular application.

2.1.5 Related Work review

The core of MPC is based on secret sharing. Dividing a secret into multiple shares in order to be able to share it with other parties was introduced by Shamir and Blakley in 1979 [11][16]. Both of these authors

worked independent of each other and they used different methods for their sharing schemes. Shamir used polynomial interpolation while Blakley used intersection of hyperplanes. Shamir's method is more well known and is used in VIFF.

MPC potentially provides solutions to many problems involving trusted third parties. After the introduction of the Millionaire Problem by Yao in 1982, some work has been done to make MPC more secure [3]. This work mainly has focused on how many adversaries a MPC can afford to have while remaining secure. In a (c,n)-threshold scheme where c represents the number of adversaries or corrupted parties and n represents the total parties, a function can be securely computed in the presence of $t < n/3$ active adversaries and $t < n/2$ passive adversaries [7][8]. Another protocol by Goldreich *et al.* presented in 1987 also ensures security of MPC in the presence of $t < n/2$ adversaries.

There has been a very limited number of MPC based applications in practice despite the fact that this technique potentially provides solutions to many problems. Although there has been work done in the developing MPC based applications, more work needs to be done in order to get the most out of MPC. There have been several frameworks developed in order to facilitate developing MPC based applications. Virtual Ideal Functionality Framework (VIFF) is an example of such a framework (it will be discussed in detail in the next section). Other frameworks include Fairplay and Sharemind [21]. Fairplay has two versions, i.e. Fairplay for two-party computations and FairplayMP for Multi-party computations (more than two parties) [19][20].

A few MPC based applications were developed using VIFF after the introduction of the framework. The first application is *Nordic Sugar*, an application for the sugarbeet contracts developed in Denmark [17]. Other applications include *Distributed RSA* [13], *Distributed AES* [18], *Ranking the Authors* and *Secure Voting* [6].

2.2 Virtual Ideal Functionality Framework

Virtual Ideal Functionality Framework (VIFF) provides a Python library allowing multiple players to execute a cryptographic protocol to do secure MPC. It works as an application prgramming interface (API) for MPC

and hides the cryptographic and communication details. VIFF handles the network communications, secret sharing, and operations on the shares,thus a developer does not need to be concerned with these details. A developer only needs to know how to use the VIFF library calls. These call will be interpreted by the Python Virtual Machine.

2.2.1 Background

VIFF, originally named PySMPC, is Python library for performing SMPC and was initially developed by Martin Geisler in 2007. PySMPC was renamed VIFF because of difficulties in pronouncing its name. As its name suggests, VIFF is a framework for developing virtual ideal functionalities. A research project named SIMAP (Secure Information Management and Processing) is considered to be the root of VIFF [22].

SIMAP's main goal was to develop tools for SMPC which can be used by normal programmers to solve real world problems without requiring that the developers be security experts. The sugarbeet contract application was the first MPC based application developed by the SIMAP project [17]. This application is based on a secure double auction which was implemented by the SCET project (Secure Computing Economy and Truct: Successor of the SIMAP project).

2.2.2 Current Features and Security Assumptions

VIFF provides an easy way to implement SMPC based applications by making use of the features provided by VIFF. Its current features as described on the VIFF webpage are discussed in this subsection [15].

The arithmetic operations discussed earlier are performed with shares for **Zp** or $\mathbf{GF}2^8$ where **Zp** and **GF** are finite fields. All the numbers involved in the SMPC are required to be from these fields in order to ensure perfect security. Secret sharing is based on Shamir's scheme and the Pseudo Random Sharing Scheme (PRSS) [27]. VIFF consists of a *field* module for handling the finite fields and separate modules for Shamir and PRSS schemes. VIFF provides overloaded operators for secure addition, subtraction, multiplication, and XORing on the shares as well as comparison of the secret shared inputs with

secret shared outputs. VIFF makes use of Twisted for asynchronous and automatic parallel execution [28]. Secure Socket Layer (SSL) is used for secure communication between two communicating parties.

VIFF documentations make it clear that the protocol is only secure if certain security assumptions are fulfilled (just like any other cryptographic system) [15]. These assumptions include that the assumption there must be majority of honest parties. The maximum number of parties that can be corrupted must be less than one third of the total number of parties in case of active adversaries and less than half of the total number of parties in the case of passive adversaries. VIFF protocols rely on assuming a certain degree of computational hardness and it is assumed that the adversary is computationally bounded, i.e., the adversary cannot overcome this computational hardness with its bounded computational power. A passive adversary observes the protocol by monitoring the traffic, but it does not inject any traffic itself.

2.2.3 VIFF architecture

VIFF consists of a number of modules for handling different features and performing various functions. The most important of these modules are the **Runtime**, **Finite Fields**, and **Shamir** modules. These modules are discussed in this section.

2.2.3.1 Runtime module

The Runtime module hides the virtual ideal functionality. Its main responsibilities are secret sharing, communication, and operations on the shares. It contains the Runtime and Share classes. The Runtime object operates on the Share object in order to perform SMPC. Every party has multiple Share objects and a single Runtime object. Each Runtime object operates on the local Share object and is connected to the Runtime objects of other parties using the ShareExchanger object. The link between two ShareExchanger objects is a SSL connection. Figure 2.2, taken from the D4.3 MPC Virtual Machine Specification (A project of CACE: Computer Aided Cryptography Engineering), shows the relationship between instances of these classes [23].

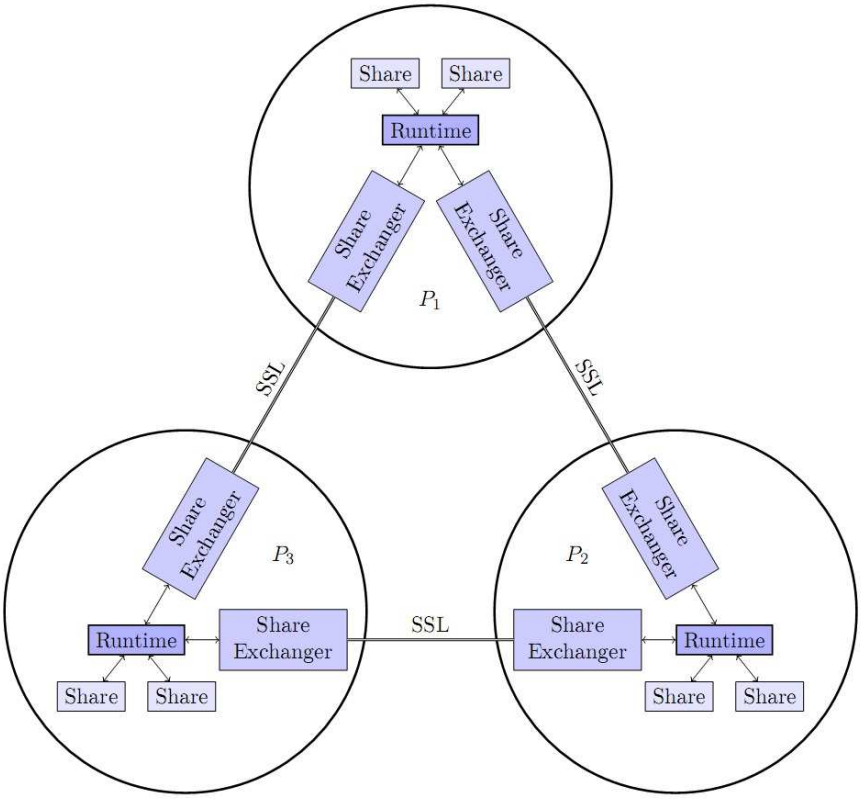

Figure 2.2: Relationship between VIFF class instances [23]

2.2.3.2 Finite Fields module

The Finite Fields module handles the finite fields elements used SMPC. It contains different classes for modeling Galois Fields. One can instantiate an object of the Galois field to get an element of the field. Operations over these elements can be performed using the overloaded arithmetic operators. The overloaded arithmetic operators are addition, subtraction, multiplication, bitwise XORing, exponentiation, etc.

2.2.3.3 Shamir module

Shamir module contains an implementation of secret sharing and recombination according to Shamir's sharing scheme. Along with other the modules,

VIFF also contains a module for Pseudo Random Sharing Scheme (PRSS) which handles secret sharing and recombination according to the Pseudo Random Sharing Scheme.

2.2.3.4 Twisted

VIFF utilizes an asynchronous design and makes use of the Twisted framework. Twisted is not an integral part of VIFF, but rather the library is used to support the asynchronous design. Twisted uses *deffered* objects to ensure asynchronous and parallel execution. A *Callback* function is called when data is available. Twisted is an event driven networking engine. Part of a function might be executed while another part might be waiting for data to be available. The total latency of a function is the sum of the local computation time, the communication time, and overhead. Local computations take much less time than distributed computations. Therefore, multiplication is more expensive than addition because addition can be performed locally. Moreover, comparison of shared secret values is very expensive because it consists of several multiplication operations.

2.2.4 MPC, VIFF, and Anonymous Authentication

The VIFF framework provides support for developing MPC based applications to solve real world problems. MPC potentially provides a solution for various problems where a trusted third party otherwise needs to be involved. MPC replaces the third party by acting as a virtual third party, hence it avoids the issues associated with a real world trusted third party. As described earlier anonymous authentication could be solved by using a trusted third party, but since MPC can be used to replace the third party, MPC can be used to solve the problem of anonymous authentication without a trusted third party. The goal of this thesis, as mentioned in Chapter 1, is to provide anonymous authentication. In the next chapter we will explain in greater details how we will use MPC and VIFF to provide anonymous authentication.

2.3 OAuth Protocol

The Open Authentication Protocol (OAuth) offers a model for authentication and authorization. OAuth introduces a third role to the traditional client-server service model. In the traditional client-server service model, a server hosts some resources owned by the client and provides other services. In the OAuth model, the resource owner is someone other than the client. The client (which is not the resource owner) requests access to the resources hosted by the server, but owned by a resource owner. The resource owner may authorize the server to allow the client to access the protected resource owned by the resource owner. A client accesses the resources of the resource owner on the server without knowing the credentials of the resource owner [2].

2.3.1 OAuth Terminology

It is important to understand the terminology used in the OAuth protocol. The terminology is given below as described in OAuth 1.0 (See RFC 5849[2]).

client An HTTP client capable of making OAuth-authenticated requests. The client requests access to some protected resources owned by the resource owner on a server.

server An HTTP server capable of accepting OAuth-authenticated requests. The server hosts the resources of the resource owner. When a client requests access to those resources, the server redirects the client to the resource owner for authorization.

protected resource The protected resources are the resources owned and controlled by the resource owner hosted on a particular server. A client can access these access-restricted resources after authentication and authorization by the resource owner using an OAuth-authenticated request.

resource owner The owner of the resources hosted on the server. The owner can access and control the protected resources by using credentials to authenticate itself with the server. It can also authorize the server to allow an authenticated client to access the protected resources.

credentials Credentials are pieces of information which allow a client to authenticate and authorize itself in order to access the resource owner's protected resources on the server. Credentials are most often a pair consisting of a unique identifier and a matching shared secret. There are three types of credentials defined in the OAuth specification. Client credentials are used to identify and authenticate the client, whereas temporary and token credentials are used to identify and authenticate the authorization request and the access grant respectively.

token When a client requests access to a protected resource on the server and the server receives authorization from the resource owner, the server issues a unique identifier, called a token, to the client which can be used by this client to associate authenticated requests with the resource owner whose authorization has been obtained by the client. Clients use a matching shared-secret in the token to establish its ownership of the token and henc authorization by the resource owner.

2.3.2 Work Flow

OAuth provides a method for clients to access server resources on behalf of a resource owner. It can help an end user by allowing a third party access to the user's resources on the server without sharing the user's credentials with the third party. The protocol is still in the process of maturing and there are several different protocol flows as explained in the OAuth RFC [2]and draft revision [24]. For the sack of illustration and simplicity, a basic protocol flow is briefly explained in this subsection. This protocol flow is called *redirection based authorization* and is coupled with the proposed model for Anonymous Authentication which is explained in the next chapter. There are other protocol flows which can also be coupled with the Anonymous Authentication system based on the requirements of the application.

The redirection based authorization method consists of three steps as specified the OAuth RFC [2]: obtaining temporary credentials, authorization, and receiving token credentials. Each of these is described below.

2.3.2.1 Obtaining temporary credentials

A client requests temporary credentials from the server in order to be able to get authorization for itself from the resource owner. Once the temporary credentials are obtained by the client, they are used as an identification for resource access request during the entire process for authorization. The server advertises a URI of an endpoint which the client can use to obtain their temporary credentials. An example URI for a temporary credentials request is something like:

```
https://www.example.net/initiate
```

A client normally has to be registered with the server with a client identifier and shared secret before it is able to request temporary credentials. The client requests the temporary credentials using the above URI. The will subsequently send these credentials as part of an access request. The server validates the request and sends a set of temporary credentials to the client in response to the request for temporary credentials.

2.3.2.2 Authorization

After obtaining temporary credentials, the client is redirected to the resource owner by the server to acquire authorization from the resource owner. The resource owner authorizes the server to allow the client access to the protected resources. The resource owner checks the authenticity of the client at this stage. The server advertises a URI for the resource owner's authorization endpoint that is used to redirect the client to the resource owner in order to be authorized by the resource owner. An example URI for an authorization request is something like:

```
https://www.example.net/authorize
```

The client redirects the resource owner to the authorization endpoint using the above URI. The server asks the resource owner for his credentials and asks him for approval of the client's request, but only if the credentials provided by the resource owner are correct. If the resource owner approves

the access request, then the client is redirected to the callback URI provided by the client in order to inform client that the resource owner has successfully completed its authorization. This confirmation also includes a *verifier* which is later used in the token request.

2.3.2.3 Token credentials

Having temporary credentials and being authorized by the resource owner, the client requests token credentials which are actually used to access the resource owner's resources on the server. The temporary credentials are revoked once the client is given the token credentials. The token credentials have limited scope and can be revoked by the resource owner at any time. The server advertises the endpoint which will be used by the client to request token credentials. An example token request URI could be:

```
https://www.example.net/token
```

The client requests token credentials using the above URI by providing its temporary credentials. The client also provides its own identifier and the *verifier*, obtained from the authorization step, along with the temporary credentials when requesting the token credentials. The server verifies all the credentials that are provided and replies with a set of token credentials.

The received token can be used to access the protected resources of the resource owner on the server. The client can make authenticated requests to ask for access to the resources. The authenticated request includes the client's identifier along with the token credentials and other parameters. The server validates the request and responds with the requested resource. The client is able to use the token to access the protected resource for a specified duration of time during which the token is valid. The resource owner can also revoke the token prematurely. In that case, the client is no longer able to use the token.

2.3.3 Prospect of coupling OAuth with MPC based authentication system

OAuth can be coupled with an MPC based authentication system in many ways according to the one of the protocol flows explained in the OAuth RFC and draft revision. The flow described fits into the authentication and authorization system when the user utilizes an OAuth client. A resource owner can be one of the authentication servers having an account on the server, providing services for each valid user. In turn the OAuth client (user) has an anonymous account on the server. The user can access this account after authenticating itself to the authentication servers and obtaining authorization from the resource owner (one of the authentication servers).

According to the OAuth protocol when the client tries to access the services provided by the server, the client cannot directly access them on its own because the service are protected. Thus the client needs obtains the temporary credentials; therefore the server redirects the client to the resource owner (one of the authentication servers) in order for the resource owner to authorize the server to allow the client to access the service. The resource owner then asks for the client's anonymous authentication. Once the client is anonymously authenticated, the resource owner authorizes the server to grant access to the client. The client then requests the token credentials. These token credentials are subsequently used by the client to access the services provided by the server.

Chapter 3

Anonymous Authentication: A Proposed Model

This chapter explains the composition of the proposed model, the relationship among the different entities that are involved in the model, the registration process, the authentication and authorization process, and the behavior of each of the individual entities in the model. The work flow of the proposed system and the behavior of individual entities are described using standard UML collaboration diagrams and SDL state transition diagrams respectively [25][26]. The sections of this chapter give a brief overview of the model before delving into the details of model.

3.1 Overview of the Model

The proposed system for anonymous authentication is composed of various interacting entities. Each of these entities plays its part at some stage in the registration, authentication, and authorization process. We will take a brief look at these individual entities before presenting the big picture.

3.1.1 Defining the Roles

Every entity or node in the system is assigned a name that indicates the role it plays. The assigned name is either the full name of the entity or a derived version of its full name.

User The *user* can be an employee of the company where the anonymous authentication system is deployed. Generally, the role of the *user* is manifest by a human user utilizing a client. This client registers itself with the system and makes use of services after authenticating itself to the system. The *user* also plays the role of an Oauth client while accessing services. The *user* registers with the *Registrar* after providing its information enabling it to register anonymously with the Authentication Servers (*CompServer*, *UnionServer*, and *GWServer*) through the *AnonAuth* (described below). The *user* accesses the services provided by the *Server* using the Oauth protocol and anonymously authenticates itself to the Authentication Servers.

Registrar The *Registrar* is one of the servers involved in the anonymous authentication system. The *user* provides its information to the *Registrar* in order to be registered. The *Registrar* validates and registers the *user* which enables the anonymous registration of the user to the other servers.

AnonAuth The *Anonymous Authenticator* (*AnonAuth*) acts a mediator between the *user* and the Authentication Servers. After successfully registering with the *Registrar*, the *user* requests the *AnonAuth* to perform anonymous registration. The request contains a secret shared and encrypted information destined for the Authentication Servers. The *AnonAuth* acknowledges the *User*'s registration after the Authentication Servers complete anonymous registration process.

CompServer and UnionServer The *CompServer* and the *UnionServer* are two of the Authentication Servers belonging to the company (or organization) and the employee union respectively. They take part in the anonymous registration and authentication process. They register the *user* after receiving the secret shared information via the *AnonAuth* and anonymously authenticate the *user* using the SMPC Authentication process. These are two of the servers, along with *GWServer*, which take part in the SMPC.

GWServer The role of the *Gateway Server* (*GWServer*) is the same as the *CompServer* and *UnionServer* during the process of registration and authentication. However, *GWServer* has some additional responsibilities, specifically it manages the *user*'s accounts on the *server* and authorizes the *server* to grant access to the *user* after it successfully authenticates itself. The *GWServer* acts as the Resource Owner of the OAuth protocol.

Server The *server* provides some services or hosts some protected resources which can be accessed by the *user* after anonymously authenticating itself to the Authentication Servers. The *server*'s role is as described in the OAuth protocol, where the *GWServer* is the resource owner and the *user* is the OAuth client.

3.1.2 The big picture

The proposed system helps the *user* to register itself, anonymously authenticate, and access the resource hosted by the *server*. We divide the complete process into two subprocesses: Registration and Authentication & Authorization. Figure 3.1 shows the basic diagram for the proposed model where the shaded part represents the Authentication Servers which take part in the SMPC to anonymously authenticate the *user*. A *user* is assumed to be registered with the system before he can access the services provided by the *server*.

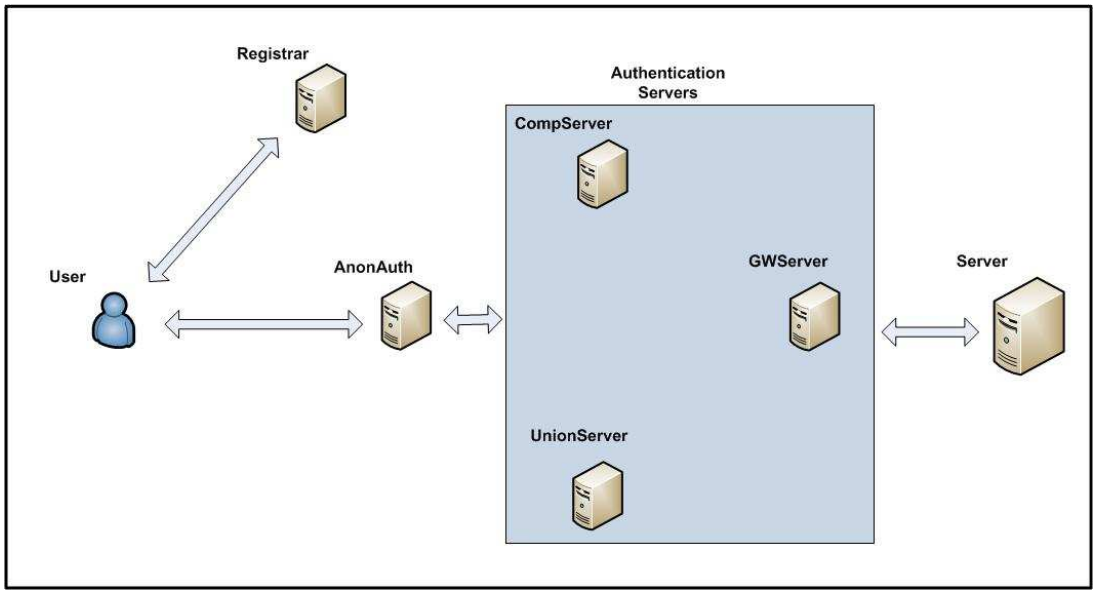

Figure 3.1: Basic Diagram of the Proposed Model

3.1.2.1 Registration process

The Registration process is initiated by the *user*. A new *User* is added to the database of the company and is provided with a one time key, **OT_Key**, which can be provided to the *Registrar* for initial registration. The *user* provides the **OT_Key** along with its other information to the *registrar*. The *registrar* validates the **OT_Key** and registers the *user*. A user's key, **U_Key**, and a one time link to *AnonAuth* is generated and provided to the *user*.

The **U_Key** is subsequently used by the *user* when performing an anonymous registration and authentication. The **U_Key** is shared using Shamir's secret sharing scheme. The shares are encrypted using the respective keys of the Authentication Servers when provided to *AnonAuth* for registration. The *AnonAuth* generates an INDEX for the Authentication Server which helps identify the *user*'s secret shared information during the authentication process. The Authentication Servers register the *User* using the provided INDEX. This INDEX is also provided to the *user*.

3.1.2.2 Authentication and Authorization

The authentication process is initiated after the *user* tries to access the resources hosted by the *server*. The *user* first obtains temporary credentials (**TempCred**) from the *server*. These temporary credentials are used as an identifier through-out the authorization process. The *server* needs authorization from the resource owner (*GWServer* in this case) to grant access to the *user*. The *GWServer* initiates authentication of *user* through *AnonAuth* before authorizing the *server* to grant access to the *user*. The *user* provides its **U_Key** in a secret shared and encrypted form to the authentication servers via *AnonAuth*.

At this point the SMPC part of the anonymous authentication occurs. The authentication servers validate the *user* by using an XOR operation on the secret shared information provided for authentication and the secret shared information provided during the registration process. The XOR operation on secret shared values is implemented by VIFF as discussed in the previous chapter. After the *user* is authenticated, the *GWServer* authorizes the *server* to grant access to the *user*. The *GWServer* also sends a **Verifier** to the *user* which helps the *user* in obtaining the token credentials (**TokenCred**). Once the **TokenCred** are obtained by the *user*, the *user* can access the protected resources on the *server* using HTTP authenticated requests until the **TokenCred** expires or is revoked by the *GWServer* [2].

3.2 Operation of the proposed system

The proposed model is explained in this section with the help of UML collaboration and choreography diagrams. The collaboration diagrams show the relationship between the different components in the model; whereas the choreography diagrams represent the occurrence of these collaborations in a specific flow. These collaborations are further elaborated using sequence diagrams showing the message flows and events that are triggered by these messages.

3.2.1 The Registration process

A *user* registers with the *registrar* and anonymously registers with the authentication server through *AnonAuth*. The *GWServer* manages the *user*'s account on the *server*. A *user*'s registration with the *registrar*, anonymous registration with the authentication servers, and the *GWServer* account management is represented by the collaborations **Registers**, **AnonRegisters**, and **ManageAcc** respectively in the collaboration diagram shown in Figure 3.2.

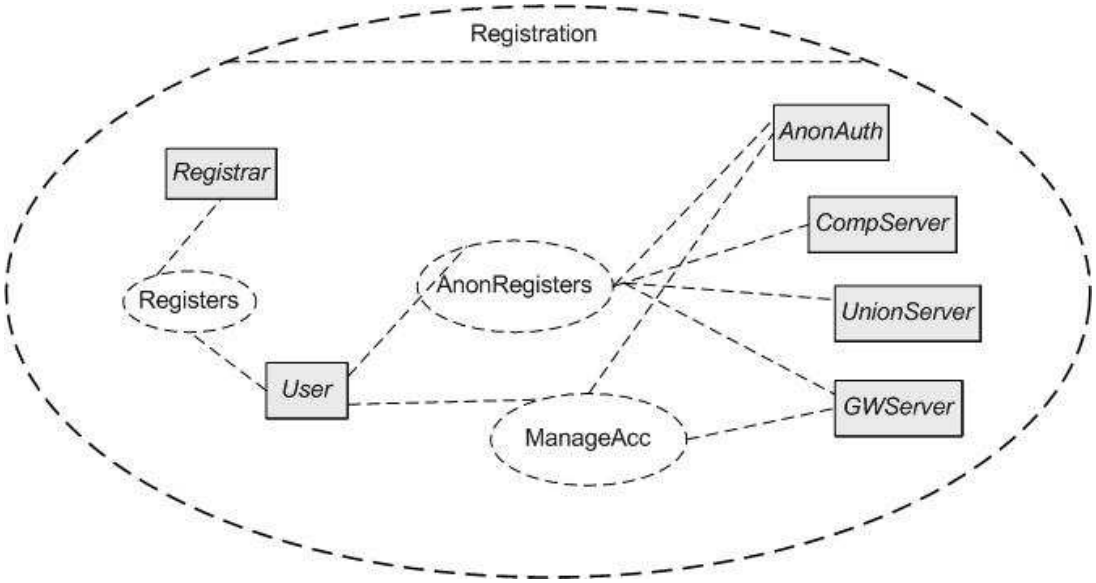

Figure 3.2: Collaboration Diagram representing the Registration process

The registration process starts with the *user* providing its information to the *registrar* in order to register. If the *user* turns out to be a valid user and the information provided is correct, then the anonymous registration process follows up. Once the *user* is anonymously registered, then the *GWServer* performs the account management tasks providing the *user* with some credentials which the *user* will require while accessing the resources on the *server*. Figure 3.3 shows the choreography or activity diagram for the registration process. The choreography diagram shows the order of the different activities and how the results from one activity might lead to different activities.

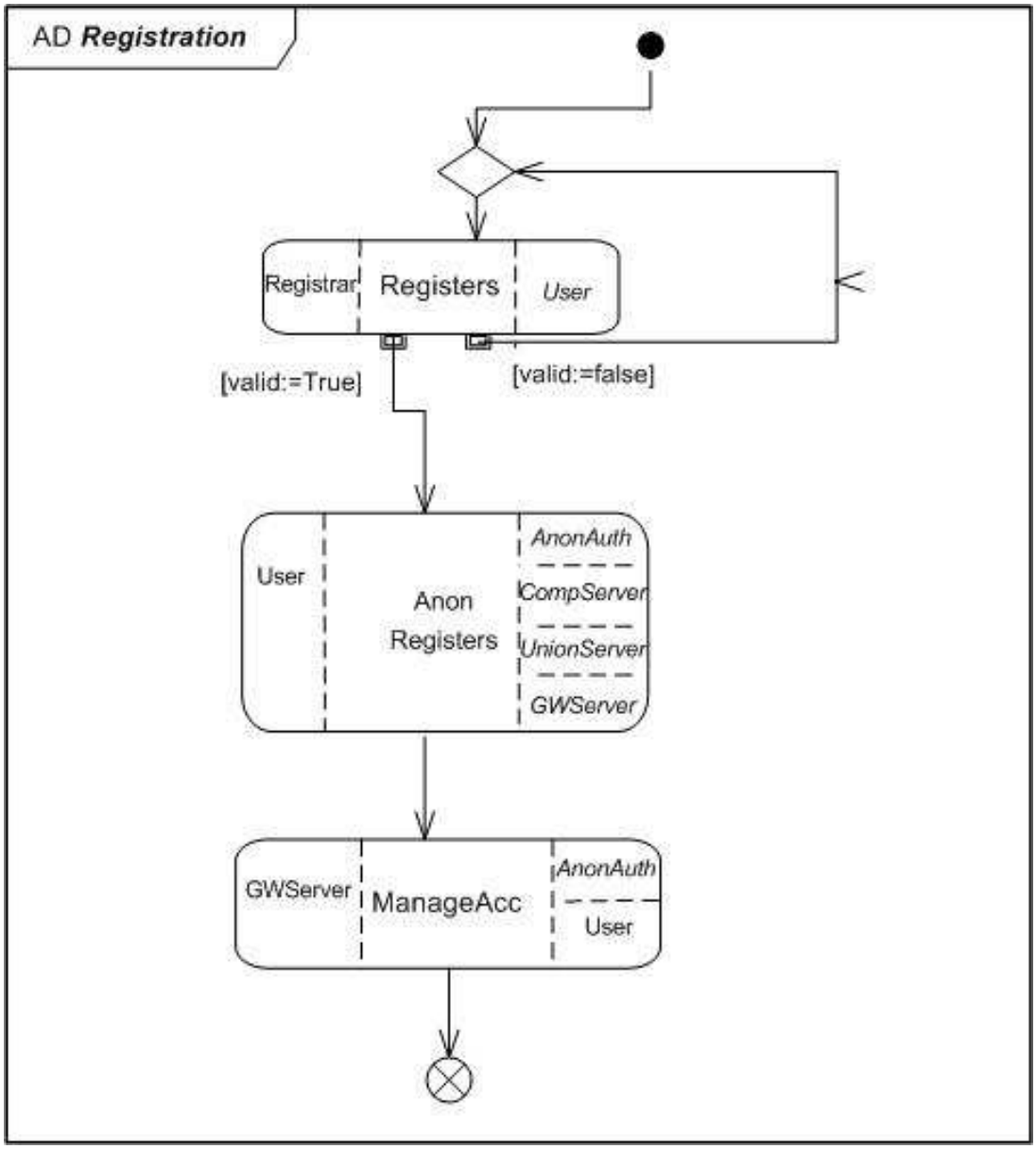

Figure 3.3: Choreography diagram for the Registration process

When a new employee joins the company, his information is added to the company's database. The new employee is also provided with a one time key, **OT_Key**, which is used to prove that he or she is a valid user when registering with the *registrar*. Figure 3.4 shows the sequence diagram explaining the collaboration **Registers**.

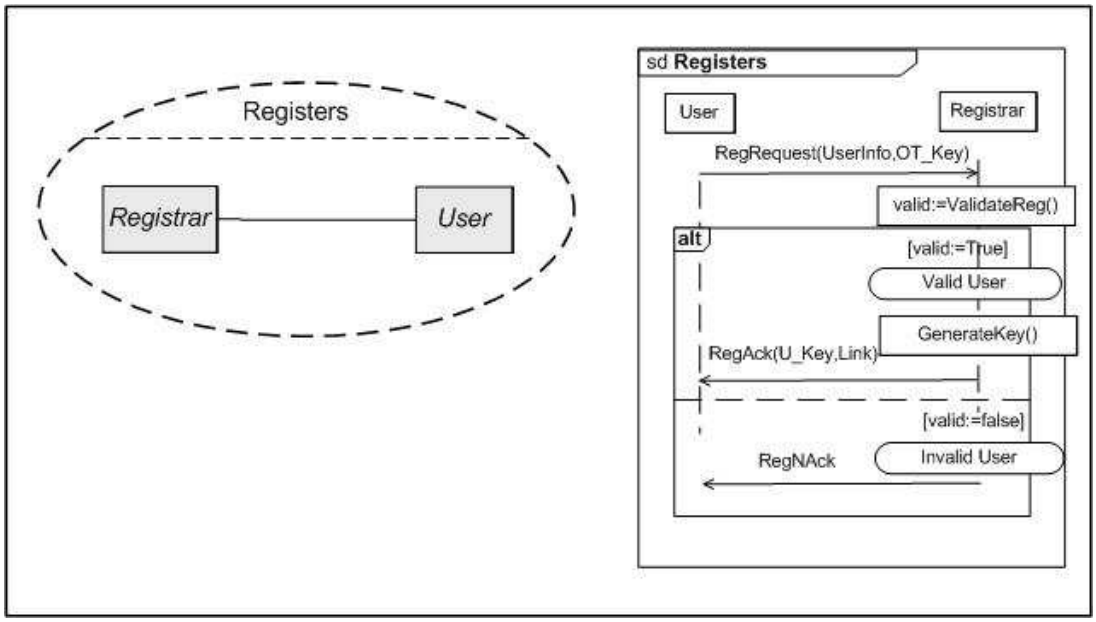

Figure 3.4: Sequence Diagram explaining the Collaboration **Registers**.

The *user* requests registration using the **RegRequest(Userinfo, OT_Key)** message. Here the **Userinfo** might contain the name of the user, position within the company, etc. The **OT_Key** is the key provided to the *user* when the *user* was added to the database for the first time and can only be used once. The *registrar* validates the information provided by the *user* and generates a pair of user key (**U_Key**) and a one time verifiable **Link** to the *AnonAuth*. If successful a **RegAck(U_Key, Link)** message is then sent to the *user*, if unsuccessful a **RegNAck** message is sent to the *user*.

A successful registration with *registrar* is followed by the anonymous registration process. The anonymous registration **AnonRegisters** is shown in Figure 3.5 using a collaboration diagram. The diagram shows that **AnonRegisters** can be divided into two subcollaborations: **AnonReg** and **RegisterUser** where the *user* only interacts with the *AnonAuth*. The *AnonAuth* handles the registration process with the authentication servers. Figure 3.6 shows the choreography diagram for anonymous registration. The diagram shows how the **AnonReg** activity leads to the **RegisterUser** activity.

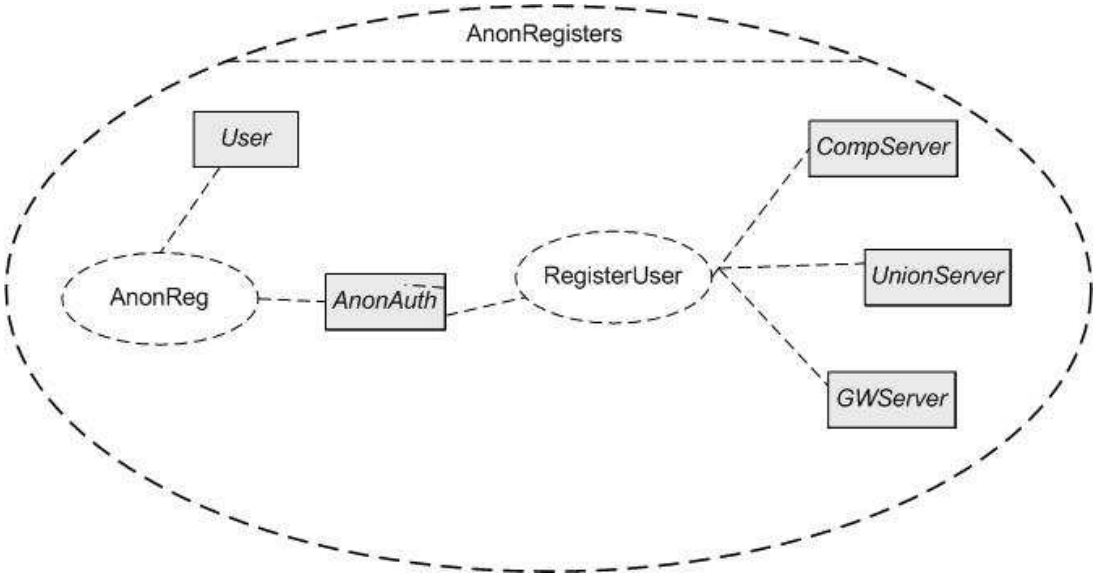

Figure 3.5: Collaboration Diagram: Anonymous Registration

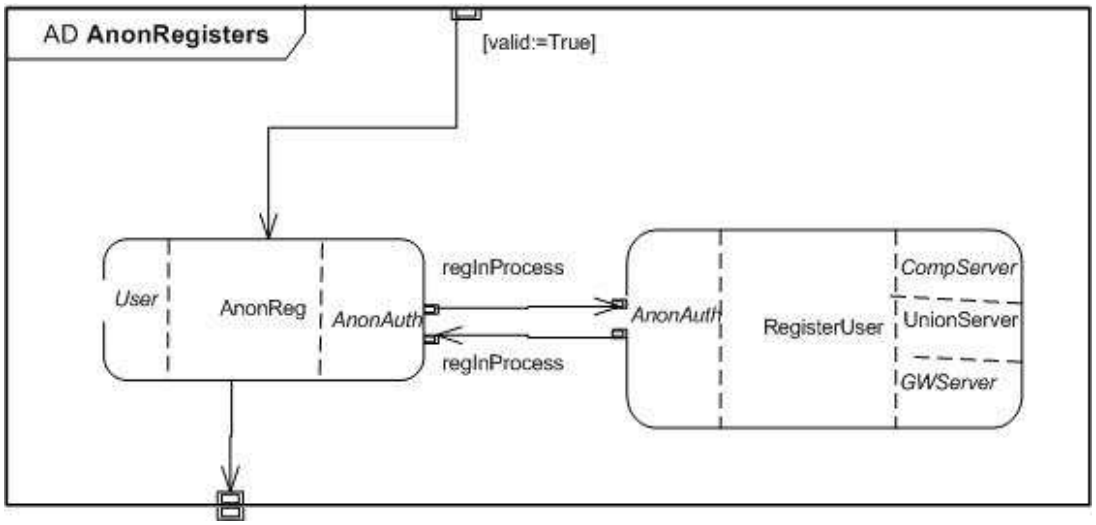

Figure 3.6: Choreography diagram for Anonymous Registration

A *user* obtains a **U_Key** and a one time **Link** to the *AnonAuth* after successful completion of registration with the *registrar*. This **U_Key** is the secret that the user shares with the authentication servers using a secret sharing scheme. The *user* creates shares of this secret **U_Key**, but encrypts it with the corresponding authentication server key when sending

it to the *AnonAuth* in the **AnonRegReq(EncShares)** message as shown the sequence diagram in Figure 3.6. The *AnonAuth* generates an **INDEX** which the authentication servers use to register the *user*. This **INDEX** later helps in the authentication process. The *AnonAuth* forwards the encrypted shares and **INDEX** to the corresponding servers in a **RegUser(Index, EncShare)** message. The authentication servers register the *user* using the given **INDEX** and send a **UserRegistered(INDEX)** to the *AnonAuth* which it forwards to the *user*.

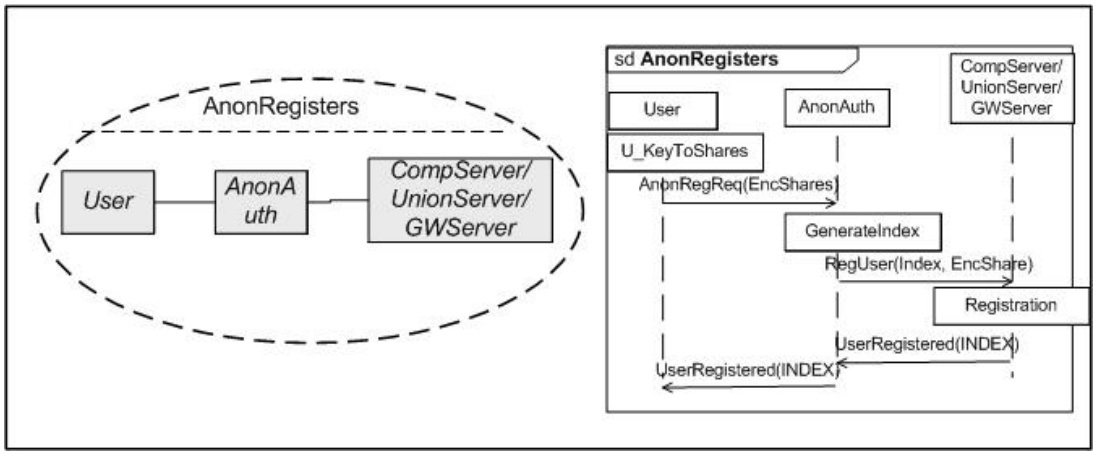

Figure 3.7: Sequence Diagram representing the Anonymous Registration

The *user* is anonymously registered with the authentication servers and can authenticate itself anonymously at a later stage. However, based on the requirements of the specific application, the *user* might need to have an anonymous account on the *server*. This account is needed to enable the *user* to see and manage his or her activities. We are using the OAuth protocol to enable the *user* access to the services provided by the *server* (where the *user* acts as an OAuth client). Here we need a resource owner who initiates the *user*'s account and authorizes the *server* to grant access to the *user*. The resource owner though must not be allowed to access the service directly from the same account. Since, the authentication process is performed by the authentication servers, the resource owner can be any of the authentication servers who have only partial knowledge of the *user*'s actual identity and therefore cannot conclude on its own whether the *user* should be given access or to determine the identity of the *user*.

We put the responsibilities of being the resource owner on the *GWServer*.

The *GWServer* has a pool of user accounts on the *server* and simply maps one of these accounts to a new *user* when the *user* registers. The *GWServer* also manages how an anonymous *user* accesses the same account every time this *user* authenticates. The sequence diagram for account management is shown in Figure 3.8.

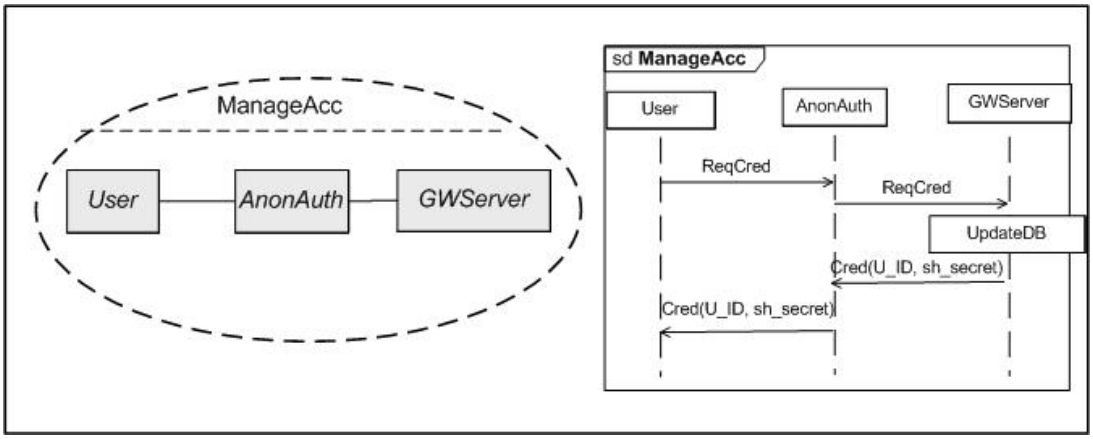

Figure 3.8: Sequence Diagram for Account Management

After a *user* is anonymously registered with the authentication servers, it requests the *AnonAuth* to provide credentials using a **ReqCred** message. This message is forwarded to the *GWServer* by *AnonAuth*. The *GWServer* maps *User* to one of the accounts from its pool of accounts and sends the necessary credentials to the *user* using the **Cred(U_ID, sh_secret)** message. Here **U_ID** and **sh_secret** are the credentials which the *user* needs to provide when requesting temporary credentials before accessing the resources on the *server*.

3.2.2 The Authentication and Authorization process

A *user* can make use of services and resources, hosted by the *server*, by accessing the account maintained for it by the *GWServer*. In order to access resources on the *server*, the *user* needs some credentials which the *server* provides after being authorized by the *GWServer*. The *GWServer* authorizes the *server* to grant access to the *user* only when the *user* is anonymously authenticated by the authentication servers.

The collaboration diagram for authentication and authorization process is shown in Figure 3.9. This complete process is composed of the sub-collaborations **AcquireTempCred**, **AnonAuthenticate**, **Authorize**, and **AcquireTokenCred**. The *user* obtains temporary credentials using the **AcquireTempCred** collaboration and authenticates itself to the authentication server using the **AnonAuthenticate** collaboration. After authenticating the *user*, the *GWServer* authorizes the *server* using the **Authorize** collaboration which allows the *user* to obtain the token credentials using the **AcquireTokenCred** collaboration.

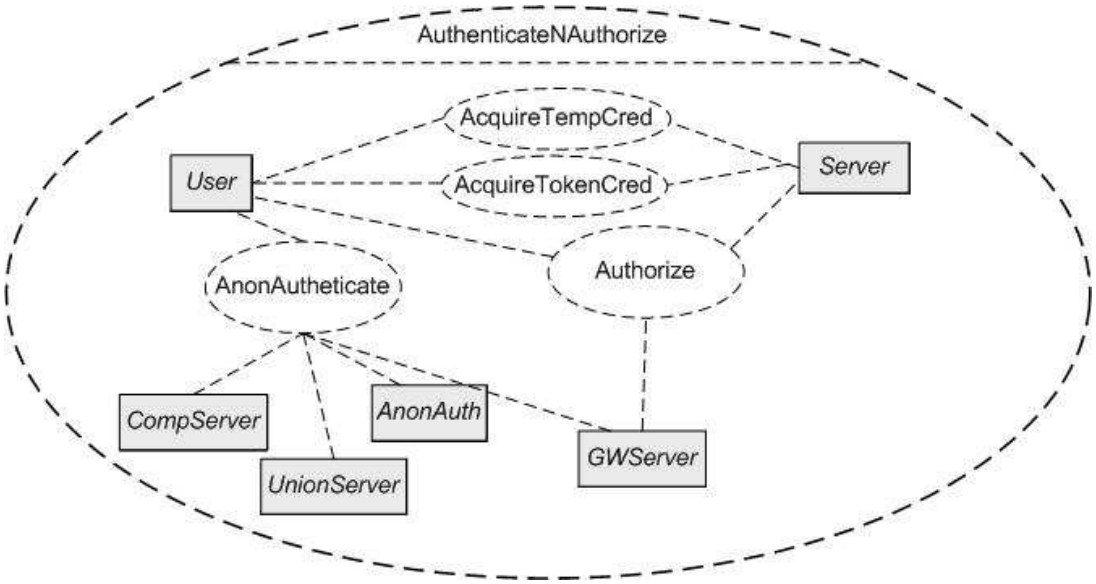

Figure 3.9: Collaboration Diagram for Authentication and Authorization

The sequence of activities is shown by the choreography diagram representing the authentication and authorization process in Figure 3.10. According to the OAuth protocol, the *user* must obtain token credentials before he or she is allowed to access the resources on the *server*. In order to be able to provide token credentials to the *user*, the *server* must seek authorization from the *GWServer*. The *GWServer* authorizes the *server* after the *user* is anonymously authenticated. The *user* obtains temporary credentials from the *server*, in the first step of this authentication and authorization process, in order to be identified throughout the process. Temporary credentials are revoked as soon as the *user* obtains token credentials.

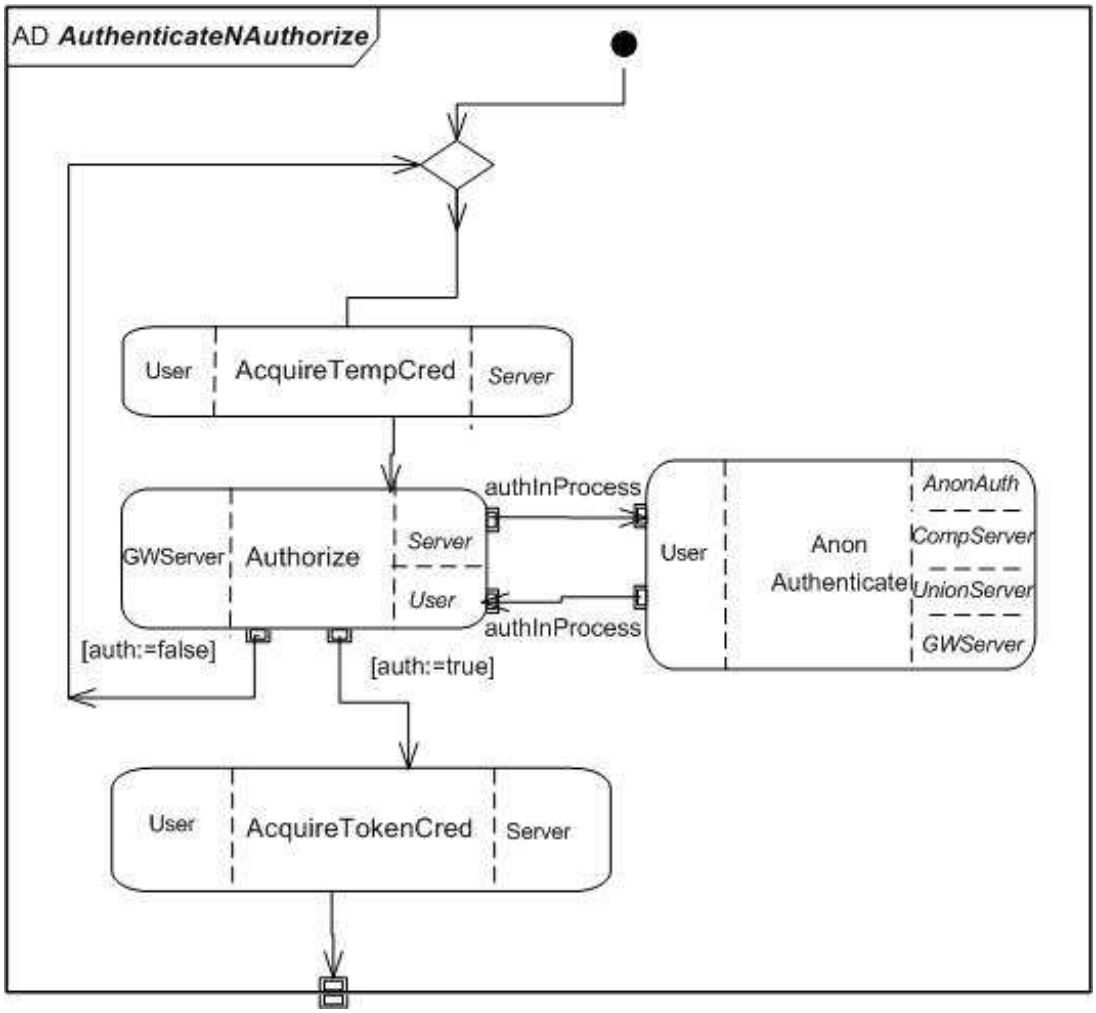

Figure 3.10: Choreography diagram representing the Authentication and Authorization process

The *user* obtains the temporary credentials using the **AcquireTempCred** collaboration whose sequence diagram is given in Figure 3.11. A **ReqTempCred(U_Cred)** message serves as a request for temporary credentials. **U_Cred** represents the **U_ID** and **sh_secret** given to the *user* at the time of anonymous registration. The *server* validates the **U_Cred** before providing the *user* with temporary credentials (**TempCred**). The requests and responses in the OAuth protocol are normally HTTP requests and responses. These message are shown in a simplified form to facilitate understanding the model and to avoid some of the irrelevant details of the original requests and responses.

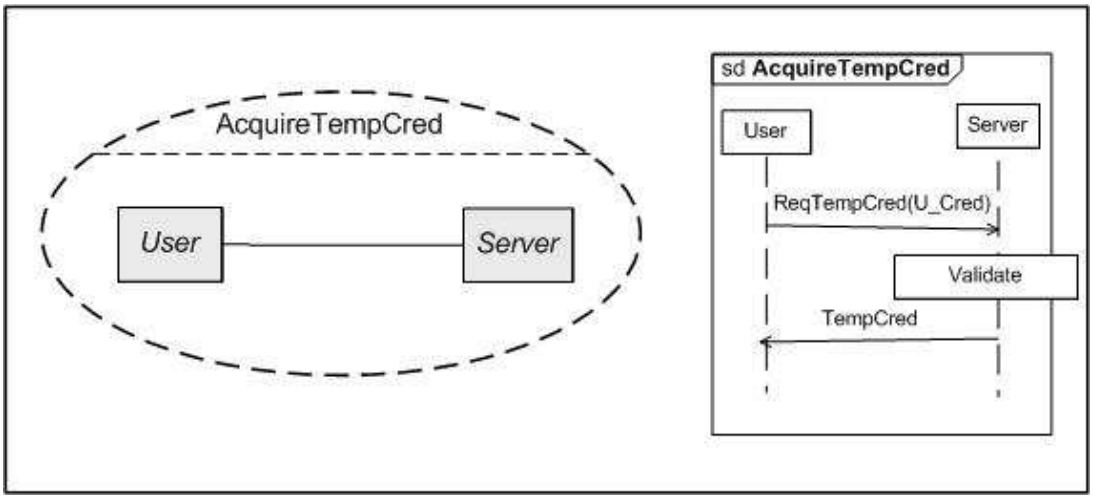

Figure 3.11: Sequence Diagram for Temporary Credentials Acquisiton

The *server* asks the *GWServer* for authorization using a **ReqAuthorize(U_Cred)** message. The sequence diagram for the authorization process is shown in Figure 3.12. The request for authorization initiates the anonymous authentication process of the *user*. The *server* is authorized using **Authorized(verifier)** message if the anonymous authentication process is successful. The *verifier*, also provided to the *user*, is later used to acquire the token credentials.

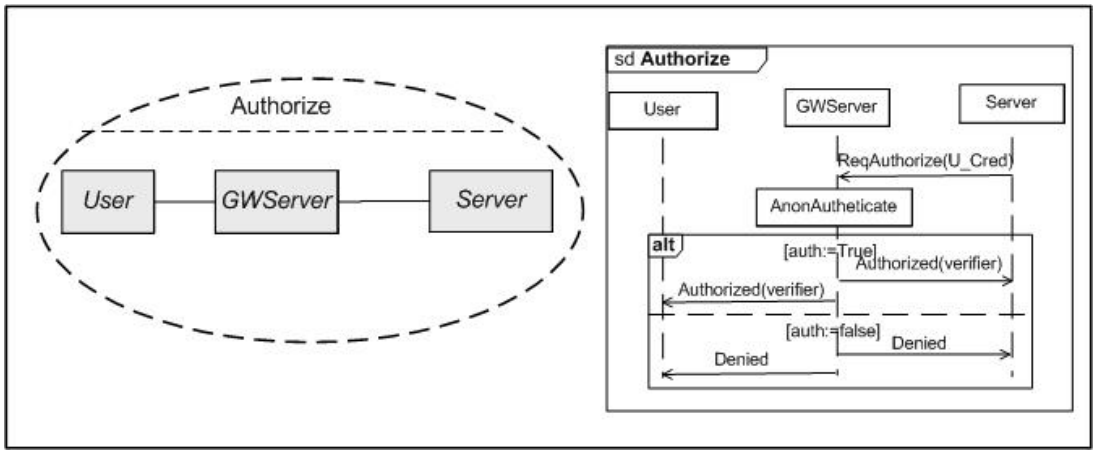

Figure 3.12: Sequence Diagram: Authorization process

The *GWServer* authorizes the *server* only after the *user*'s anonymous

authentication was successful. The anonymous authentication process is shown using a sequence diagram in Figure 3.13. This process is initiated by the *GWServer* with an **InitiateAuth** message to the *AnonAuth* which is subsequently forwarded to the *user*. The *user* responds with a **ReqAuthenticate(Index, Shares)** message to the *AnonAuth*, where Index is the *INDEX* used by the authentication servers to register the *user* and **Shares** are the shares of the secret **U_Key** encrypted with the corresponding keys of the authentication servers. The *AnonAuth* sends an **AuthenticateUser(Index, Share)** message to each of the authentication servers containing the index and share destined for that particular server. The *user* is anonymously authenticated by comparing the current secret shared value with the secret shared value provided during the registration process. This is done using the XORing of secret shared values based on SMPC. XORing of secret shared values is performed by VIFF.

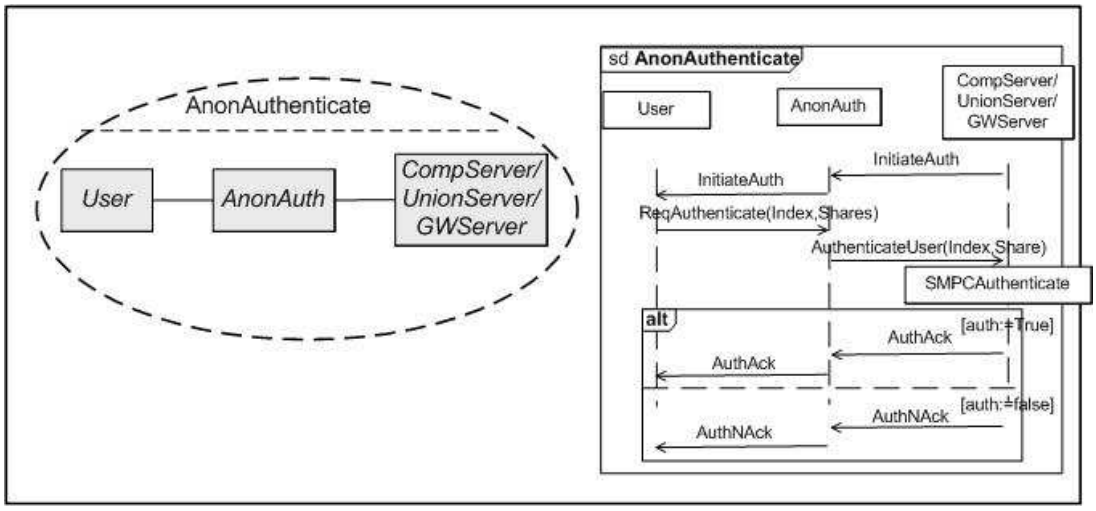

Figure 3.13: Sequence Diagram: Anonymous Authentication

An acknowledgment message is sent to the *user* and the *server* is authorized to grant access to the *user* if the anonymous authentication ends successfully. At the end of a successful authorization process, a *user* gets a **verifier** which it uses to obtain the token credentials. Acquisition of the token credentials (**TokenCred**) is shown in Figure 3.14 using a sequence diagram. The *user* requests the **TokenCred** using the **ReqTokenCred(U_Cred, TempCred, verifier)** message. **U_Cred** identifies the *user*, **TempCred** binds the session, and **verifier** proves that this is the *user* for whom the *server* is authorized to grant access to the resource. The *server* validates

all this provided information and responds with the **TokenCred**. The *User* can now access the resources with an HTTP authenticated request using the **TokenCred** until the **TokenCred** either expires or is revoked by the *GWServer*.

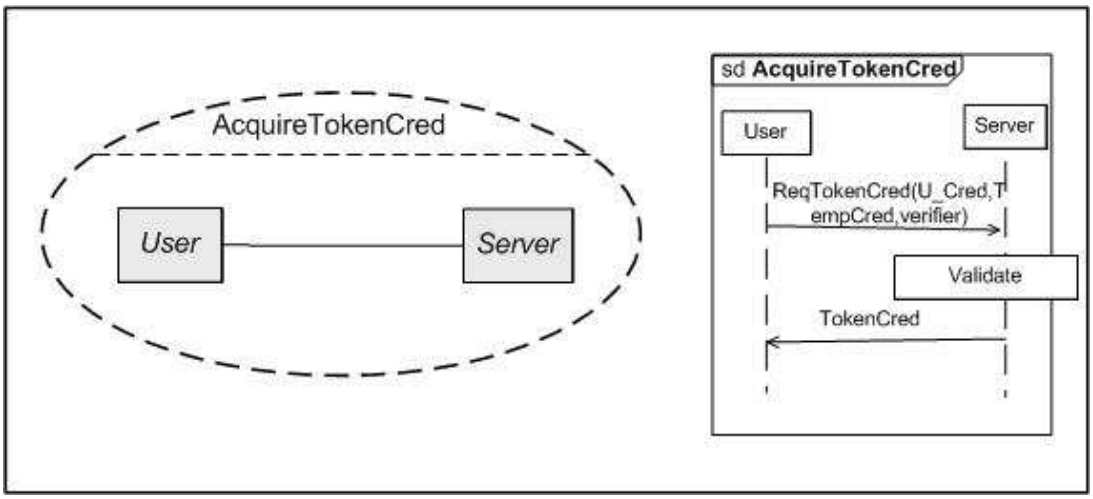

Figure 3.14: Sequence Diagram: Acquiring Token Credentials

3.3 Behavior of the individual entities

This section focuses on the behavior of individual components or entities in different circumstances. We examine their behavior one by one through a thorough explanation of the proposed model. Their behavior is explained using SDL state transition diagrams. These state transition diagrams shows the transition of a component from one state to another state along with the activity triggered by a specific input signal. The following subsections explain the behavior of the individual components.

3.3.1 Registrar

The *registrar* handles the initial registration of the *user*. The SDL state transition diagram for the *registrar* is shown in Figure 3.15. The *registrar* is in the **idle** state until it receives a **RegRequest** message from the *user*. After receiving the message it validates the information provided by the *user*

in the request. If the *user* is a valid user, it generates the **U_Key** and a one time **Link** to the *AnonAuth* and sends these to the *user* along with a **RegAck** message. In the case of an invalid user, it sends a **RegNAck** message to *user*. The *registrar* remains in **idle** state in both the cases.

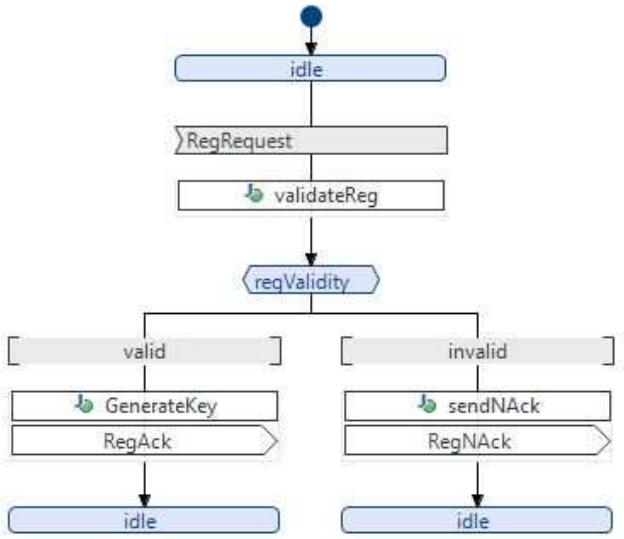

Figure 3.15: SDL State Transition Diagram for the *Registrar*

3.3.2 AnonAuth

AnonAuth takes part in both the registration process as well as authentication. The SDL state transition diagrams for the *AnonAuth* are shown in Figure 3.16 and Figure 3.17. *AnonAuth*'s role in the registration process starts when it receives an **AnonRegReq** message from the *user* when in the **idle** state. The *AnonAuth* generates an **INDEX** and sends a **RegUser** message to authentication servers along with the shares of the secret information in encrypted form provided by the *user*. The *AnonAuth* moves to the **regInProcess** states and waits there until it receives a **UserRegistered** message from the authentication servers. It forwards the received message to the *user* and moves to the **accountManage** state. In this state it waits for the **ReqCred** message from the *user*. When it receives this message, it forwards to the *GWServer* and moves to the **finalizeReg** state. When it receives credentials from the *GWServer* and forwards these

credentials to the *user*, then it moves back to the **idle** state.

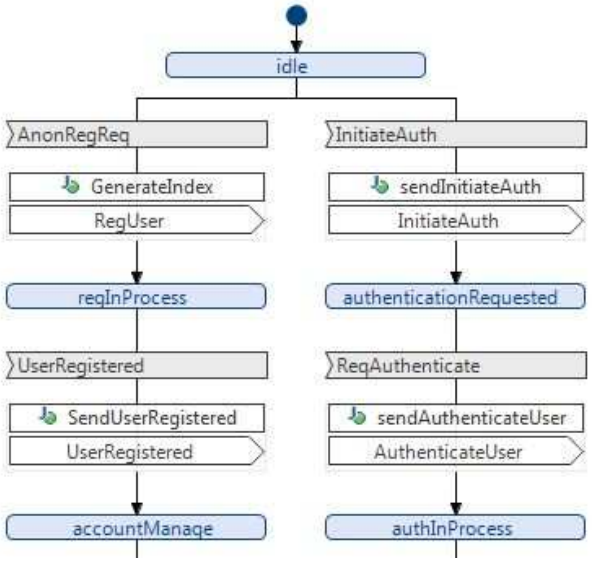

Figure 3.16: SDL State Transition Diagram1: *AnonAuth*

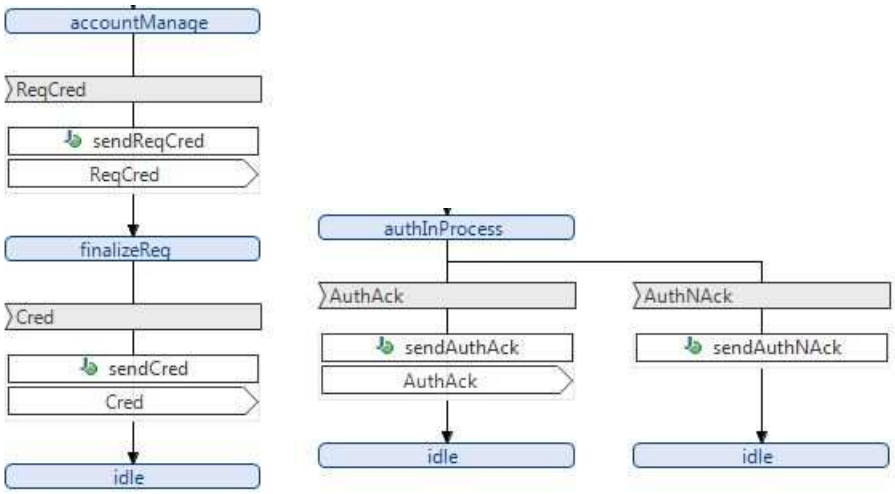

Figure 3.17: SDL State Transition Diagram2: *AnonAuth*

If the *AnonAuth* receives an **InitiateAuth** message from the *GWServer* while in the **idle** state, it forwards this message to the *user* and moves to the **authenticationRequested** state. When it receives a **ReqAuthenticate**

message along with an **INDEX** and shares of the secret in encrypted form, then it forwards the **INDEX** and the corresponding secret shares to the authentication servers in an **AuthenticateUser** message and moves to the **authInProcess** state. In this state, it either receives an **AuthAck** or an **AuthNAck** message depending upon whether the *user* is valid user or not. In both the cases, it forwards the message to the *user* and moves back to the **idle** state.

3.3.3 CompServer and UnionServer

The *CompServer* and *UnionServer* behave in the same manner both during registration and authentication. Therefore their SDL state transition diagrams are the same (as shown in Figure 3.18). If they receive a **RegUser** message from the *AnonAuth* along with an **INDEX** and an encrypted share of a secret value, they register the *user* using the provided **INDEX** and send a **UserRegistered** message to *AnonAuth* while remaining in the **idle** state.

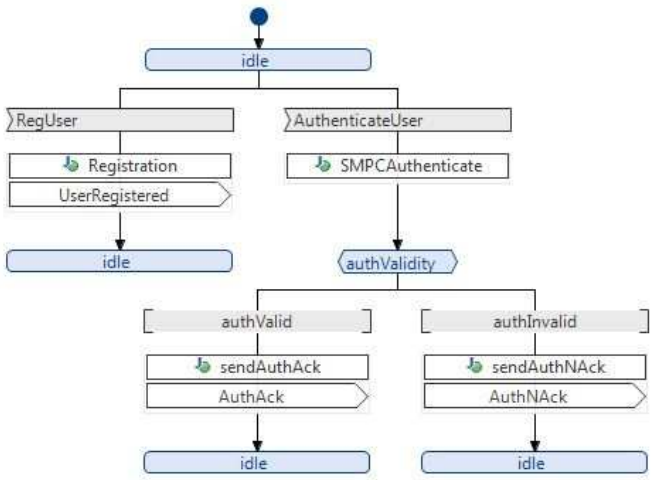

Figure 3.18: SDL: *CompServer* and *UnionServer*

They can receive an **AuthenticateUser** message from the *AnonAuth* while in the **idle** state during the authentication process. This message contains an index and a share of the secret value. They perform an *XOR* using SMPC of the current secret shared value with the secret shared value stored

together with the given index. The *user* is successfully authenticated if both the secret shared values match. These servers send an **AuthAck** or **AuthNAck** message to *AnonAuth* depending upon the result of the authentication process and will remain in the **idle** state.

3.3.4 GWServer

The behavior of the *GWServer* is more or less the same as the *CompServer* and *UnionServer* except for the fact that it is also responsible for some account management tasks. The SDL state transition diagram for the *GWServer* is shown in Figure 3.19. When the *GWServer* registers the *user*, it moves to the **accountManage** state rather than returning back to the *idle* state, unlike the other authentication servers. It waits for a **ReqCred** message from the *user* through the *AnonAuth*. When it receives the **ReqCred** message, it performs the necessary account management tasks and sends the credentials to the *user* via *AnonAuth* before moving back to the **idle** state.

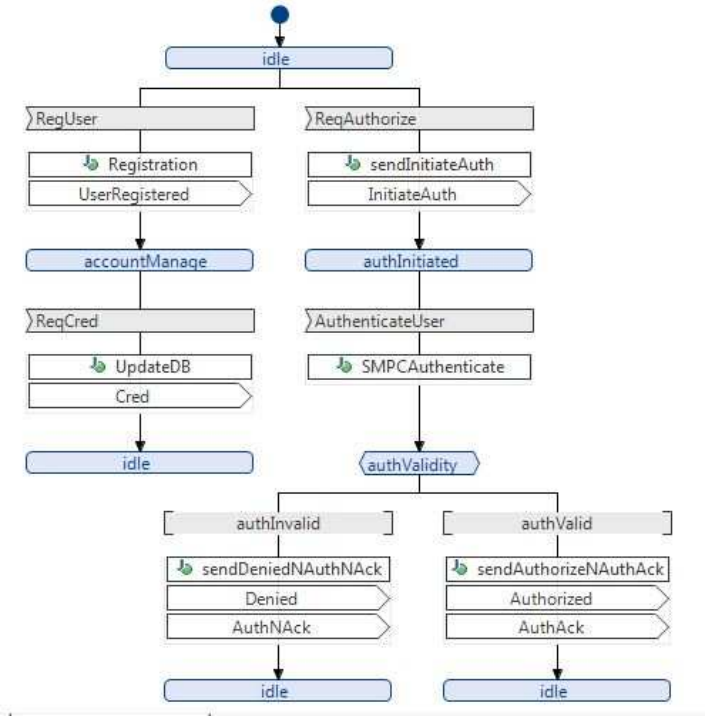

Figure 3.19: SDL State Transition Diagram: *GWServer*

When a *user* tries to access some resources on the *server*, the *server* sends a **ReqAuthorize** message to the *GWServer*. While in the **idle** state, when the *GWServer* receives this message, it sends an **InitiateAuth** message to *AnonAuth* and moves to the **authInitiated** state. When it receives an **AuthenticateUser** message in this state, it performs the SMPC based authentication process just as the other authentication servers. The only difference is that it also sends an **Authorized** or **Denied** message to the *server* along with sending the **AuthAck** or **AuthNAck** message to the *user* (depending upon the success of the authentication process). It moves back to the **idle** state in both the cases.

3.3.5 User

A *user* is the most important component of the whole system and is involved in all the processes. The *user*'s SDL state transition diagrams are shown in Figure 3.20 and Figure 3.21. A *user* is the initiator of both the registration and authentication process. The *user* sends a **RegRequest** message, along with the necessary information for registration, to the *registrar* and moves to the **unregistered** state. When it receives a **RegNAck** message in the case of an invalid *user*, it remains in the **unregistered** state. In the case of a valid *user*, it receives a **RegAck** message along with a **U_Key** which it provides in a secret shared and encrypted form enclosed in an **AnonRegReq** message before moving to the **halfRegistered** state. When it receives a **UserRegistered** message in this state, it requests the credentials using a **ReqCred** message and moves to the **credWait** state. When it receives the credentials in this state, it becomes eligible to access the resources on the *server*.

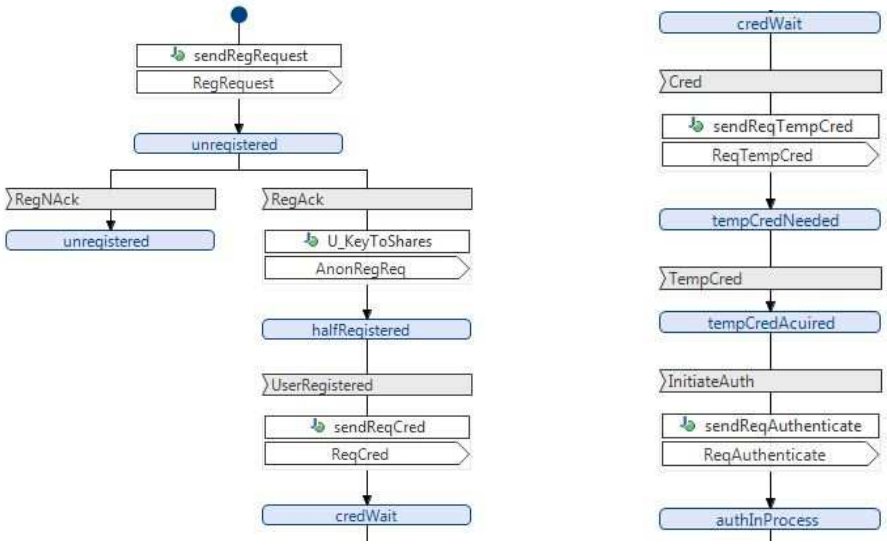

Figure 3.20: SDL State Transition Diagram: *User*

In order to access the resources on the *server*, it requests temporary credentials from the *server* and moves to the **tempCredNeeded** state until it receives the **TempCred**, at which it moves to the **tempCredAcquired** state. At this stage, the *server* requires authorization from the *GWServer* and the *GWServer* needs to authenticate the *user*. To do this the *GWServer* sends an **InitiateAuth** message to the *user* through the *AnonAuth*. The *user* responds with a **ReqAuthenticate** message along with the necessary information and moves to the **authInProcess** state. An **AuthNAck** message in this state moves the *user* back to the **tempCredAcquired** state where the *user* needs to repeat the authentication process once again. In contrast reception of an **AuthAck** message prompts the *user* to request for the token credentials and move to the **tokenCredNeeded** state. When in this state it will wait to receive the **TokenCred**, then it moves to the **authenticated** state. At this point the *user* can access the resources on the *server* until it receives an **InvalidToken** message which means that the token has either expired or been revoked by the *GWServer*. Upon receiving an **InvalidToken** message, the *user* asks for temporary credentials and has to pass through the authentication process again.

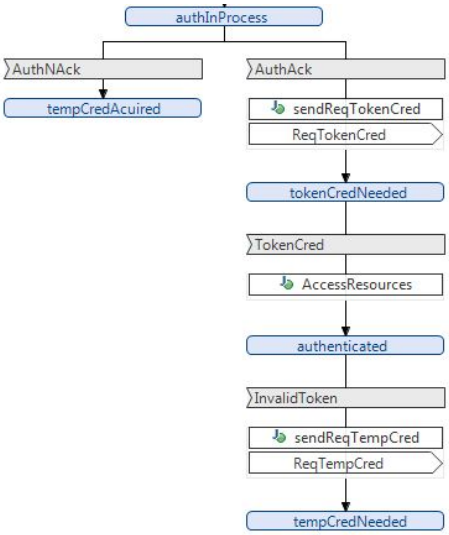

Figure 3.21: SDL State Transition Diagram2: *User*

Chapter 4

Security Analysis of the proposed model

This chapter provides a theoretical analysis of the model proposed in the previous chapter. The main goal of this thesis project was to design a system by which a user can authenticate itself in such a manner that none of the servers, involved in the authentication process, can individually know the actual identity of the user. In other words, the user's authentication must be anonymous and only a valid user should be able to successfully complete the authentication process.

4.1 Issues to be addressed

The problem of anonymous authentication is dealt with using SMPC in the proposed design. There are certain basic issues that need to addressed before this anonymous authentication system can be implemented. The proposed system is based on SMPC. SMPC needs to be sufficiently secure that a limited number of adversaries can not compromise the security of the system or affect the end result in any way. Another issue that needs to be addressed is the anonymity of the user. How is the anonymity of a user assured? How can we be sure that the user will be anonymous, but still able to access the same resources every time the user authenticates?

Security of the complete system is one of the major concerns. An invalid user must not be able to authenticate or access protected resources on the user's behalf. In fact no one should be able to access the account created for a particular anonymous user except the user. The following sections of the chapter discuss these issues and try to indicate how these issues were handled during the design of the proposed model. Possible areas of further improvement are also discussed at the end of this chapter.

4.2 Secure Multi-party Computation

Secure Multi-party Computation (SMPC) potentially provides a solution to problems which are generally solved by involving a trusted third party. A trusted third party normally takes inputs from the parties involved in the computation, performs some arithmetic operations on the inputs according to a specified function, and reveals the output of the computation. The trusted third party guarantees the correctness of the output and that the inputs are not revealed to any one. SMPC replaces this trusted third party in order to avoid concerns with using a trusted third party as discussed in chapter 1.

Since SMPC replaces the trusted third party, it must also guarantee the same properties that were guaranteed by the trusted third party. Security in SMPC requires the correctness of the output and esuring the secrecy of the individual inputs even if some parties try to cheat [1]. SMPC operate successfully despite some bounded number of adversaries involved in the MPC, while still producing a correct output and maintaining the secrecy of the inputs. According to Ben-Or, Goldwasser, & Widgerson and Chaum, Crpeau, & Damgrd, SMPC provides perfect security for computing any function if the number of corrupt parties is less than $n/2$ where n is the total number of parties in the MPC [7][8]. Since the number of servers involved in the design can be adjusted according to the situation, this number of servers can be fixed in such a way that none of the parties can control more that half of the servers and hence no one will be able to compromise the security of the system. SMPC also assumes that there is an unconditionally secure channel between every two parties. This secure channel is ensured by using SSL in VIFF.

The framework used for realizing SMPC is VIFF. VIFF makes some security assumptions which must be in place to assure that the system is secure. The

security assumptions are based on the security requirements of SMPC. The assumptions say that the adversary can corrupt a number of parties less than a certain threshold. The threshold is assumed to be half of the parties, thus implying that there must be an honest majority in the participating parties. The adversary is also assumed to be computationally bounded. If all these assumptions and constraints are met, the SMPC can considered secure.

4.3 Anonymity of the user

The concept of MPC is to distribute information in such way that none of the participating parties has all the information. MPC provides a method of performing arithmetic operations on the distributed secret shared information in a such a manner that the original information is not revealed to any one. In our design, the goal was to hide the identity of the user so that the identity is not revealed during authentication. This implies that none of the servers involved in the authentication must posses all the information about the user's identity. However, any number of servers greater than the threshold can share their information and obtain the actual identity of the user.

Our design makes it sure the user's actual identity is only provided to the *registrar* when the user is registered for the first time. In all of the subsequent registration or authentication processes only a user key **U_Key**, provided by the *registrar*, is used. The **U_Key** is not used in its actual and plain text form, but rather in the form of encrypted shares of the secret. The shares are encrypted with the corresponding keys of the authentication server. So, no one can decrypt the shares other than the corresponding authentication server. Even if all the authentication servers agree to reveal the identity of user, they will not be able to do so even with their combined effort. They will only be able to reconstruct the **U_Key** which does not tell anything about the user's actual identity. To learn the user's actual identity, they will require the services of the *registrar* to match the **U_Key** with the entries in its database in order to lookup the user's actual identity. Thus the user's actual identity is protected from all server except for the *registrar* and the *registrar* is not involved in authentication.

4.4 Security of the system

The proposed model assures the anonymity of the user, but this is of no use if the overall system is insecure. Our proposed model ensures that an invalid user is not be able to register, authenticate successfully, or access the protected resources on the user's behalf. The model also ensures that no one, except for the users themselves are able to access the anonymous account created for the user.

The registration process is secured by the introduction of a one time key **OT_Key** provided to every new user when the user is first added to the user database. The user needs to provide this **OT_Key** along with other information to the *registrar* at the time of initial registration. Subsequently anonymous registration commences only if the user's initial registration with the *registrar* was successful. This ensures that only valid users can be registered.

An invalid user cannot authenticate to system because this invalid user would have to successfully pass through the authentication system and provide the secret index as well as the shares of the secret **U_Key**– however, this information can only be provided by a valid user. Even if the adversary eavesdrop one session, they will not gain sufficient information to be able to authenticate an invalid user in another session. Timestamps and nonces ensures that each session is fresh one.

However, the *GWServer* posses the information about the account and could authenticate itself and access the services itself. This security weakness can be detected by maintaining a secure log of users who were authenticated by the system and a log of users who accessed the resources. The *GWServer* obviously will be able to access the resources, but it cannot authenticate itself to the system by pretending to be a user. The *GWServer* can access the protected resources on the server because the resources are owned by the *GWServer*. However, to access the protected resources pretending to be a valid user, the *GWServer* would have to provide the index and shares of the secret **U_Key**. Since, the *GWServer* can not provide the shares of the the secret **U_Key**, the *GWServer* cannot access the protected resources pretending to be a valid user. This ensures no one other than the user can access the resources on user's behalf.

4.5 Areas that need further improvement

This chapter described solutions to most of the concerns, but although some of these solutions are expected to be trivial to implement, they are not yet incorporated in the proposed model. The proposed model will only be completely secure if all of these solutions are included in the design before implementing and testing it. Timestamps and secure logs are two such features which must be included in the design to ensure security or to detect the *GWServer* by-passing the system's access controls. Both of these features are included in the future plan for the project, but were not considered essential to implement at this stage of the project.

The other area that needs some improvement is the OAuth protocol. As mentioned earlier, the protocol flow used in the proposed design is a very basic one and may not be the most optimal one. However, the choice of flow depends upon specific situations and further research needs to be done to find the optimal (i.e., most efficient) protocol flow for different specific situations.

Chapter 5

Development of the System

SMPC is a very new field of research in terms of application development and we could not find much work done in the area of anonymous authentication. We believe that designing and implementing a solution for anonymous authentication exceeds the scope of a masters thesis project. However, a model is proposed as a solution for the problem as explained and analyzed in the earlier chapters. Implementation and testing of the proposed model, however, will require additional time and effort. Although we have managed to built a skeleton of the proposed model, this skeleton is the output of the SDL state transition diagrams of chapter 3. It contains all the messages that are exchanged between different components of the system. However, there is still a need to realize the functionality of each of the necessary functions. The rest of this chapter explains how these functions can be realized.

5.1 SMPC part of the design: VIFF development

All the components involved in the system are not part of SMPC. Only *user*, *CompServer*, *UnionServer*, and *GWServer* are components involved in SMPC. Since VIFF is a Python library each of these components needs to have a few lines of Python code to realize it semantics. This code can be invoked from the java code of the overall system at appropriate locations. For example, a *user* needs to create shares of the secret input and can use

the method given in Listing 5.1 to do so. This method is based on Shamir's secret sharing scheme.

```
1
   //VIFF implemented Method for creating shares.
3
   shares = rt.shamir_share([1, 2, 3], GF256, input)
```

Listing 5.1: VIFF method for creating shares

The authentication servers need to compare two secret shared values in order to find out the whether the *user* is a valid user or not. They can learn this by *XOR*ing the two values. They can also learn this by using a subtraction operation. Listing 5.2 shows the VIFF method for *XOR*ing two secret shared values *a* and *b*.

```
2  //VIFF implemented Method for XOR-ing two secret shared
       numbers a and b.
4  xor(self, share_a, share_b)
```

Listing 5.2: VIFF method for XORing of two secret shared values a and b

5.2 Skeleton of the System

This section discusses the important functions of the components and briefly describes how the corresponding methods can be completed with the appropriate code.

5.2.1 Registrar

A *registrar* receives registration requests and acknowledges the *user*'s requests based on the validity of this *user*. However, its main function is to validate the *user* and generate the **U_Key** and one time verifiable Link to the *AnonAuth*. Listing 5.3 provides the two methods that need to realize these functionalities. A *user* is initially validated by validating the *user*'s provided user information and the **OT_Key** by comparing it with values in the user database.

```
2   //Method for validating the user

4   public static void validateReg(RegRequest signal, RegistrarSM
        asm){
        // Code for validating a user's registration request
6   }

8   //Method for Generating the U_Key and the one time Link to the
        AnonAuth

10  public static void GenerateKey(RegistrarSM asm){
        // Code for generating the U_Key and one time Link to
            AnonAuth
12  asm.sendMessage(new RegAck(), " address ");
    }
```

Listing 5.3: Important functions of the *Registrar*

5.2.2 AnonAuth

AnonAuth is a mediator between the *user* and the authentication servers. Its job is, most of the time, to forward the received messages back and forth to the appropriate destination. However, its own function is to create a unique **INDEX** which is used by the authentication servers to register the *user*. Listing 5.4 provides a method for this **INDEX** generation which needs to be completed with the appropriate code.

```
1
    //Method for generating INDEX against which a user is
        registered in the Authentication Servers
3
    public static void GenerateIndex(AnonRegReq signal, AnonAuthSM
        asm){
5       // Code for generating the INDEX
    asm.sendMessage(new RegUser(), " address ");
7   }
```

Listing 5.4: Important function of the *AnonAuth*

5.2.3 CompServer and UnionServer

The functionality of the *CompServer* and *UnionServer* is the same in both the registration and authentication processes. They register the *user* by storing the provided information in their databases and authenticate the user by the comparing the newly provided information with the information provided during the registration process. The methods for both the registration and authentication are given in Listing 5.5. Since, authentication is performed using SMPC, the method responsible for authentication should be linked with the Python code for SMPC.

```
1
   //Method for the user's registration
3
   public static void Registration(RegUser signal, CompServerSM
       asm){
5       // Code for registering a user
   asm.sendMessage(new UserRegistered(), " address ");
7  }

9  //Method for the SMPC authentication

11 public static void SMPCAuthenticate(AuthenticateUser signal,
       CompServerSM asm){
       // Code for invoking the Python code for SMPC
           authentication.
13 }
```

Listing 5.5: Important functions of the *CompServer* and *UnionServer*

5.2.4 GWServer

Listing 5.6 shows the methods representing the important functionalities of the *GWServer*. The functionality of the *GWServer* is the same as that of the *CompServer* and the *UnionServer* during a *user*'s registration and authentication. However, the *GWServer* also has some extra account management responsibilities. The **UpdateDB()** method in Listing realizes the account management functionality and should be completed accordingly.

```
1
   //Method for user's anonymous registration
3
```

```
public static void Registration(RegUser signal, GWServerSM asm
    ){
    // Code for user's registration
asm.sendMessage(new UserRegistered(), " address ");
}

//Method for Account Management

public static void UpdateDB(ReqCred signal, GWServerSM asm){
    // Code for Account management tasks
asm.sendMessage(new Cred(), " address ");
}

//Method for the SMPC authentication

public static void SMPCAuthenticate(AuthenticateUser signal,
    GWServerSM asm){
    // Code for invoking the Python code for SMPC
        authentication.
}
```

Listing 5.6: Important functions of the *GWServer*

5.2.5 User

Modern day design of the Internet is pushing the complicated part of the applications towards the core of network and edges are made as simple as possible. The *user* in our design is also not very complicated. It performs only one key function along with sending requests to various servers and receiving their responses, it that it creates shares of the secret **U_Key** and provides these shares to the authentication servers in encrypted form both in registration as well as during the authentication process. The method shown in Listing 5.7 should depend upon some Python code to create shares of the secret value.

```
//Method for generating the shares of secret U_Key and
    encrypting them

public static void U_KeyToShares(RegAck signal, UserSM asm){
    // Code for invoking the python code to create shares of
        U_Key
asm.sendMessage(new AnonRegReq(), " address ");
}
```

Listing 5.7: Important functions of the *User*

Chapter 6

Implementation of a simplified prototype

We have implemented a simplified prototype of the proposed model which realizes the basic registration and authentication process. The prototype consists of four entities: *user*, *compServer*, *unionServer*, and *gwServer*. The *user* shares a **U_Key** with these authentication servers in a secret shared form and then tries to authenticate using the same **U_Key**. The authentication servers perform a SMPC comparison operation using the VIFF libraries. At the end of the authentication process, each of the authentication servers knows whether the authentication was successful or not. Each entity in the prototype operates only for a single registration and authentication process. All the entities shut down on completion of the authentication process irrespective of the result of authentication. The prototype is implemented in Python. Functionalities of the *user*, *compServer*, *unionServer*, and *gwServer* are realized by `user.py`, `compServer.py`, `unionServer.py`, and `gwServer.py` respectively. The Python code for these files is given in the appendix.

6.1 Generating the configuration files and starting the entities

Each of the entities involved in SMPC has a configuration file. This configuration file contains the keys and network information corresponding to all the parties involved in SMPC. A party involved in SMPC is also called a *player*. So, there is a configuration file for every player. These configuration files are generated before the start of SMPC. We ran the authentication servers and the user on the same computer. The network address of each player is specified while generating the configuration files. The configuration files for four player, running on the same computer, are generated using the following command.

```
python generate-config-files.py -n 4 -t 1 localhost:9001\
localhost:9002 localhost:9003 localhost:9004
```

This command runs the file `generate-config-files.py` which can generate configuration files for any number of players. The network addresses of player-1, player-2, player-3, and player-4 are `localhost:9001`, `localhost:9002`, `localhost:9003`, and `localhost:9004`. Each of the player is configured according to its configuration file. Configuration files for player-1 is given in the appendix.

Each of these players needs to be started. These players can be started in any order. When the player is started, the configuration file is given as an argument to it. A player waits for the other players when it is started. All the four players are started in separate terminals using the following commands.

```
python user.py player-1.ini
python compServer.py player-2.ini
python unionServer.py player-3.ini
python gwServer.py player-4.ini
```

6.2 Registration

The *user* is asked to provide a **U_Key** when the `user.py` starts. The *user* provides the **U_Key** and waits for the completion of registration with the authentication servers as shown in Figure 6.1(a snapshot of the *user*'s screen during registration). The **U_Key** is shared with the authentication servers using Shamir's sharing scheme.

```
Enter your U_Key to be registered with the Servers: 12345
Not using SSL
Listening on port 9001
Will connect to <Player 2: localhost:9002>
Will connect to <Player 3: localhost:9003>
Will connect to <Player 4: localhost:9004>
```

Figure 6.1: User waiting for completion of registration

The *user* is notified when the authentication servers complete the registration process as shown in Figure 6.2. Operation of all the authentication servers is the same. These authentication server start and then wait for the other authentication server to start. When all of the these authentication servers are started, they receive shares of the **U_Key** and register the *user* as shown in Figure 6.3(a snapshot of the compServer's screen only).

```
Enter your U_Key to be registered with the Servers: 12345
Not using SSL
Listening on port 9001
Will connect to <Player 2: localhost:9002>
Will connect to <Player 3: localhost:9003>
Will connect to <Player 4: localhost:9004>
You are registered now!!! :)
```

Figure 6.2: User notified after completion of registration

```
Listening on port 9002
Will connect to <Player 3: localhost:9003>
Will connect to <Player 4: localhost:9004>
User is registerd!!!
```

Figure 6.3: A snapshot of compServer's screen when a user registers

6.3 Authentication

After completion of the registration process, the *user* is asked to provide to the **U_Key** in order to authenticate. The *user* provides the **U_Key** which is shared with the authentication servers using Shamir's secret sharing scheme. The authentication servers validate the **U_Key** using SMPC. The *user* is notified of a successful authentication if the **U_Key** provided by the *user* is correct as shown in Figure 6.4. A snapshot of the compServer's terminal, after successful authentication of the *user*, is given in Figure 6.5.

```
Enter Your U_Key for authentication: 12345
You are authenticated!!!
Synchronizing shutdown... done.
Closing connections... done.
Stopping reactor... done.
```

Figure 6.4: User notification after successful authentication

```
Authentication Successful
Synchronizing shutdown... done.
Closing connections... done.
Stopping reactor... done.
```

Figure 6.5: A snapshot of compServer's terminal after user's successful authentication

If the **U_Key** provided by the *user* in not valid, the *user* is notified of the denial of authentication as shown in Figure 6.6. The authentication servers display an **Authentication Failed** message, if the **U_Key** provided by the *user* is invalid, as shown in Figure 6.7. The *user* and the authentication servers shut down after both successful and unsuccessful completion of the authentication process.

```
Enter Your U_Key for authentication: 23456
Authentication Denied!!!
Synchronizing shutdown... done.
Closing connections... done.
Stopping reactor... done.
```

Figure 6.6: User's unsuccessful authentication

Figure 6.7: Authentication Failed

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Traditional authentication systems authenticate the user after the user proves his or her valid identity. Based on the requirements of the service, it might be important that the user should be able to authenticate without his or her identity being revealed to anybody; not even the authentication server. This leads to a problem called the Anonymous Authentication problem. The idea behind this problem is that a valid user must be able to authenticate and access protected resources without revealing his or her actual identity to any one.

A solution to these sort of problems is generally a trusted third party solution, in this thesis project we have replaced the trusted third party with Secure Multi-party Computation (SMPC) due to issues that occur when using an actual trusted third party. A model for anonymous authentication was developed based on SMPC by using VIFF as a development framework. The anonymous authentication system is coupled with the OAuth protocol to allow the user to access protected resources. A java based skeleton of the proposed model was developed which still needs to filled in to realize appropriate functionalities. A simplified prototype, realizing a basic anonymous authentication system, of the proposed model was implemented. This prototype was developed in Python using VIFF libraries to realizes SMPC.

While there has been very little work done in SMPC regarding application development, this thesis project provides a potential base for anonymous authentication. This base could potentially lead to many projects based on anonymous authentication and therefore can practically serve as the pioneering work in this field.

7.2 Future Work

The proposed model requires some refinements and addition of some new features as mentioned at the end of chapter 4. The proposed system could then be implemented using the code skeleton and tested in some real world scenarios. Once, the anonymous authentication system has been developed, it could be coupled with various services to provide anonymity.

Appendix A

Skeleton of the proposed model

A.1 SMPC part of the model

A.1.1 User

```
from optparse import OptionParser
import viff.reactor
viff.reactor.install()
from twisted.internet import reactor

from viff.field import GF
from viff.runtime import create_runtime, Runtime
from viff.config import load_config

parser = OptionParser()
Runtime.add_options(parser)
(options, args) = parser.parse_args()

Zp = GF(1031)

id, players = load_config(args[0])

input1 = input('Enter your U_Key to be registered with the
    Servers: ')

def protocol(rt):

    def got_result(result):
```

```
23          if result == 0:
                print "You are authenticated!!!"
25          else:
                print "Authentication Denied!!!"
27          rt.shutdown()

29      x1, y1, z1, w1 = rt.shamir_share([1, 2, 3, 4], Zp, input1)

31      print "You are registered now!!! :)"

33      input2 = input('Enter Your U_Key for authentication: ')

35      x2, y2, z2, w2 = rt.shamir_share([1, 2, 3, 4], Zp, input2)

37      diff = x1 - x2
        opened_diff = rt.open(diff)
39      opened_diff.addCallback(got_result)

41
pre_runtime = create_runtime(id, players, 1)
43 pre_runtime.addCallback(protocol)

45 reactor.run()
```

Listing A.1: user.py

Configuration files for all the parties are similar to the one given in the below listing.

```
1 # VIFF config file for Player 1

3 [Player 1]
    host = localhost
5   port = 9001
    [[paillier]]
7     type = viff
      [[[pubkey]]]
9       g =
566782711799011092641459335709964309365332967734756414461
11 089915452360059382528237666305469439356231206718051077821

    1590987181050434013376790951002636882791365118182379003
479868116644524262781340504363674885393092005019446037 9
13 141361072541536761531357411324885772883913812656349242 3
735931892324216977010695032818119666275591911623268950 6
15 404452460385467032492227153468748919621272890497891363 2
826754406766746564459995355364843373732731673397788822 5
17 409533872887839689841090306773989400054788222402869517 8
105333680085582311038764032311340855837553109435241502 4
```

```
19 9352245704237827416406010108482520473597020724461052295
   86935215
21       n =
   262458689422056035634283987812027234688057661744535709 5
23 1146605230919231211730165724670857481839040972656810686
   0445766402663575503773987569873263986705195899027977400
25 1531609395196782736801748973689493468962801312935673727
   4521091962371732844477063188252384031241633948907257442
27 1221903690510718565911369324651 51
     [[[seckey]]]
29       lm =
   1312293447110280178171419939060136173440288308722678547
31 5573302615459615605865082862335428740919520486328405343
   0222883201331787751886993784936631993352597875639710174
33 6275597550150810053740646127243540389922594147923142022
   8736826278110429392507771155432720057460562078027075335
35 6865327881168783173684130302569 52
       g =
37 5667827117990110926414593357099643093653329677347564144
   6089915452360059382528237666305469439356231206718051077
39 8215909871810504340133767909510026368827913651181823790
   0347986811664452426278134050436367488539309200501944603
41 7914136107254153676153135741132488577288391381265634924
   2373593189232421697701069503281811966627559191162326895
43 0640445246038546703249222715346874891962127289049789136
   3282675440676674656445999535536484337373273167339778882
45 2540953387288783968984109030677398940005478822240286951
   7810533368008558231103876403231134085583755310943524150
47 2493522457042378274164060101084825204735970207244610522
   9586935215
49       n =
   262458689422056035634283987812027234688057661744535709 5
51 1146605230919231211730165724670857481839040972656810686
   0445766402663575503773987569873263986705195899027977400
53 1531609395196782736801748973689493468962801312935673727
   4521091962371732844477063188252384031241633948907257442
55 1221903690510718565911369324651 51
     [[prss_keys]]
57     1 3 4 = 0x1a62b461b17e5028fd35f1e89d0e68112b1055efL
       1 2 3 = 0x9aeaad1a98673bf7338cba7902cc33fd9962f295L
59     1 2 4 = 0xc57cf71f316c6cd43cc11ec3cd562893f40620d9L
     [[prss_dealer_keys]]
61     [[[Dealer 1]]]
         1 3 4 = 0x9bf83e70e0c108ab4d34671ac797f0f0b5e0053eL
63       1 2 3 = 0x6a928718c30853fe1f72bcf7f3c4c64ee9a3e4b6L
         2 3 4 = 0x1fa66a196f6065654782e15cf86e07a44407c825L
65       1 2 4 = 0x431473a17f94dfe7c9680148f1a3e1706a1c8dd8L
       [[[Dealer 2]]]
67       1 3 4 = 0x304b4c3a2fa4d911038556512c3fea1e9f34f9b6L
```

```
      1 2 3 = 0xbf2d04368e1be935368cfb0d5a8abdb312ed89c0L
      1 2 4 = 0xf6d9441664c98ddd65aa7bc89fbd3d0cc28ff76cL
    [[[Dealer 3]]]
      1 3 4 = 0xa7b47fc69c639c672d94ce436880b695b7079240L
      1 2 3 = 0xda0cc02084a2ebe1042b31429c6f3aa0e999087L
      1 2 4 = 0x39926fe2fc99e0f0454a924d59c9f219963f580L
    [[[Dealer 4]]]
      1 3 4 = 0x59ac11c2cc0b3c31b9272e34b5dec018e330493eL
      1 2 3 = 0x6e1bd58b3cdb3b578915a2c39dc0ee88b6b99a5eL
      1 2 4 = 0x38f92062204f3a8e6ad0d93315bcc57ab58f4d08L

[Player 2]
  host = localhost
  port = 9002
  [[paillier]]
    type = viff
    [[[pubkey]]]
      g =
14535318449013166346169507677104860896595690760666992252651837761012440462811949464793739290288531259549252145467920699473110420700179001113592474999584501739801061627706482821430447202040962804140480975655825625560205791574946971987332638088983229797847188218189773570032804680656601203754636747024360704294568742832906883202648356459070245674921364749468110352703987067355076467769832642617194883443248695436099430488180307095127667945950320191388491186427801978842945976311946354209547374196885108308854945929449296431151947310535390498784051401829203883977310300743489867651812780373710120534646887017397547697
      n =
28537057501262453845873773399048537123631216337413930790676164434278610894726746170506408401227714569305989949090128453068299758301317071024262792224320476250372716512758964857357927395349368093163458113140688111502892024980863867441908706023072081758671410745924191953387924551694633978595710472723032050315
[Player 3]
  host = localhost
  port = 9003
  [[paillier]]
    type = viff
    [[[pubkey]]]
      g =
96063216852800873834094005814189629457136202550142892390480535698583895379104760547620372865009240837878686047124084507259124316460969037425957580645086724045863912670155657166472168409078677163049689091313024770702813281456511634466204240651262690592381003356786515858020725804824736980587701877
```

```
244097860476403168415692740999744688843912170534064924331587777518521549652104628997595999090227051313719271067336431653323168311432880284146294704090568621467848240560956281851542281007579630756565862931319865506011292549533797380444102421684606850645479045618453614446369555415377638296957324617366405855854019822619175
      n =
100297929818274169551863075470924703910585832969668077262669273666550776070529132755078037576402086833177445616197818276901084967075076771283004599279964601878863873625115609582806994726929294843423952964071080740293213426177136705890753462018093608282731537132558325425897087959178743873593889544344423231235
      g =
775836136985234802117671915788159120616452380798124274885405912805605303721380446309211390394429387145667947645121193778317968081170026549897881224504107814348457248151560778982222787613476557568195786376843459658018826444057899310929086442829803393407055510840306659847902627879977779934832739150948943876635266746730067566159950397208753739621943227749992843353162127214127546638489066652165350345469550880298954576688092419312021711672563324760413702994134585703128408509348407929160834475719343687987520059054101325115510699318186137168733480142464765477486071883249162486593963239675756072305805760676862492755
      n =
503269373296975524869928097876207246125676376015452724486294485651185387842034338713760232077436327167124552595521572284325833758322916462220392830918833866846206146955982010607450720016070690609248999503271550287285079786726812835281270325270740388271790066003339105705617066260651622634563911317967611248833

# End of config
```

Listing A.2: player-1.ini

A.1.2 Authentication Servers

The behavior of all the authentication server is the same in SMPC part of the model. As an example, compServer.py (python file for the company server) is given in the below listing.

```
from optparse import OptionParser
import viff.reactor
viff.reactor.install()
from twisted.internet import reactor

from viff.field import GF
from viff.runtime import create_runtime, Runtime
from viff.config import load_config

parser = OptionParser()
Runtime.add_options(parser)
(options, args) = parser.parse_args()

Zp = GF(1031)

id, players = load_config(args[0])


def protocol(rt):

    def got_result(result):
      if result == 0:
          print "Authentication Successful"
        else:
          print "Authentication Failed"
        rt.shutdown()

    x1, y1, z1, w1 = rt.shamir_share([1, 2, 3, 4], Zp, 0)

    print "User is registerd!!!"

    x2, y2, z2, w2 = rt.shamir_share([1, 2, 3, 4], Zp, 0)

    diff = x1 - x2
    opened_diff = rt.open(diff)
    opened_diff.addCallback(got_result)


pre_runtime = create_runtime(id, players, 1)
pre_runtime.addCallback(protocol)

reactor.run()
```

Listing A.3: compServer.py

A.2 Registrar.java

```
package anonymousauthentication.registrar;

import anonymousauthentication.RegRequest;
import anonymousauthentication.RegNAck;
import anonymousauthentication.RegAck;


public class RegistrarActions extends Object {

  public static void validateReg(RegRequest signal, RegistrarSM
      asm){
       // User's registration
  }

  public static boolean valid(RegRequest signal, RegistrarSM asm
      ){
       return false;
  }

  public static boolean invalid(RegRequest signal, RegistrarSM
      asm){
       return false;
  }

  public static void sendNAck(RegistrarSM asm){
       asm.sendMessage(new RegNAck(), " address ");
  }

  public static void GenerateKey(RegistrarSM asm){
       // Generating U_Key and one time Link to AnonAuth
  asm.sendMessage(new RegAck(), " address ");
  }

}
```

Listing A.4: Registrar.java

A.3 AnonAuth.java

```
1  package anonymousauthentication.anonauth;

3  import anonymousauthentication.AnonRegReq;
   import anonymousauthentication.RegUser;
5  import anonymousauthentication.UserRegistered;
   import anonymousauthentication.ReqCred;
7  import anonymousauthentication.Cred;
   import anonymousauthentication.InitiateAuth;
9  import anonymousauthentication.ReqAuthenticate;
   import anonymousauthentication.AuthenticateUser;
11 import anonymousauthentication.AuthAck;
   import anonymousauthentication.AuthNAck;
13

15 public class AnonAuthActions extends Object {

17   public static void GenerateIndex(AnonRegReq signal, AnonAuthSM
         asm){
         // Generating the INDEX
19   asm.sendMessage(new RegUser(), " address ");
     }
21
     public static void SendUserRegistered(UserRegistered signal,
        AnonAuthSM asm){
23       asm.sendMessage(new UserRegistered(), " address ");
     }
25
     public static void sendReqCred(ReqCred signal, AnonAuthSM asm)
        {
27     asm.sendMessage(new ReqCred(), " address ");
     }
29
     public static void sendCred(Cred signal, AnonAuthSM asm){
31       asm.sendMessage(new Cred(), " address ");
     }
33
     public static void sendInitiateAuth(InitiateAuth signal,
        AnonAuthSM asm){
35       asm.sendMessage(new InitiateAuth(), " address ");
     }
37
     public static void sendAuthenticateUser(ReqAuthenticate signal
        , AnonAuthSM asm){
39       asm.sendMessage(new AuthenticateUser(), " address ");
     }
41
```

```
    public static void sendAuthAck(AuthAck signal, AnonAuthSM asm)
        {
43  asm.sendMessage(new AuthAck(), " address ");
    }
45
    public static void sendAuthNAck(AuthNAck signal, AnonAuthSM
        asm){
47  asm.sendMessage(new AuthNAck(), " address ");
    }
49
}
```

Listing A.5: AnonAuth.java

A.4 CompServer.java and UnionServer.java

Since CompServer.java and UnionServer.java are exactly the same, so only CompServer.java is included here.

```
package anonymousauthentication.compserver;
2
import anonymousauthentication.RegUser;
4 import anonymousauthentication.UserRegistered;
  import anonymousauthentication.AuthenticateUser;
6 import anonymousauthentication.AuthAck;
  import anonymousauthentication.AuthNAck;
8

10 public class CompServerActions extends Object {

12   public static void Registration(RegUser signal, CompServerSM
         asm){
       // User's registration
14   asm.sendMessage(new UserRegistered(), " address ");
     }
16
     public static void SMPCAuthenticate(AuthenticateUser signal,
         CompServerSM asm){
18       // SMPC authentication process. Python code for SMPC is
             invoked from this method
     }
20
     public static boolean authValid(AuthenticateUser signal,
         CompServerSM asm){
22       return false;
```

```
    }
24
    public static void sendAuthAck(CompServerSM asm){
26  asm.sendMessage(new AuthAck(), " address ");
    }
28
    public static boolean authInvalid(AuthenticateUser signal,
        CompServerSM asm){
30      return false;
    }
32
    public static void sendAuthNAck(CompServerSM asm){
34  asm.sendMessage(new AuthNAck(), " address ");
    }
36
}
```

Listing A.6: CompServer.java

A.5 GWServer.java

```
1  package anonymousauthentication.gwserver;

3  import anonymousauthentication.RegUser;
   import anonymousauthentication.UserRegistered;
5  import anonymousauthentication.ReqCred;
   import anonymousauthentication.Cred;
7  import anonymousauthentication.ReqAuthorize;
   import anonymousauthentication.InitiateAuth;
9  import anonymousauthentication.AuthenticateUser;
   import anonymousauthentication.Authorized;
11 import anonymousauthentication.AuthAck;
   import anonymousauthentication.Denied;
13 import anonymousauthentication.AuthNAck;


15
   public class GWServerActions extends Object {
17
     public static void Registration(RegUser signal, GWServerSM asm
         ){
19       // User's Registration
     asm.sendMessage(new UserRegistered(), " address ");
21   }

23   public static void UpdateDB(ReqCred signal, GWServerSM asm){
       // Code for User's Account Management
```

```
25    asm.sendMessage(new Cred(), " address ");
      }
27
      public static void sendInitiateAuth(ReqAuthorize signal,
          GWServerSM asm){
29    asm.sendMessage(new InitiateAuth(), " address ");
      }
31
      public static void SMPCAuthenticate(AuthenticateUser signal,
          GWServerSM asm){
33        // SMPC authentication process. Python code for SMPC is
              invoked from this method
      }
35
      public static boolean authValid(AuthenticateUser signal,
          GWServerSM asm){
37        return false;
      }
39
      public static boolean authInvalid(AuthenticateUser signal,
          GWServerSM asm){
41        return false;
      }
43
      public static void sendAuthorizeNAuthAck(GWServerSM asm){
45          asm.sendMessage(new Authorized(), " address ");
          asm.sendMessage(new AuthAck(), " address ");
47      asm.sendMessage(new Denied(), " address ");
    asm.sendMessage(new AuthNAck(), " address ");
49  }

51  public static void sendDeniedNAuthNAck(GWServerSM asm){
      asm.sendMessage(new Denied(), " address ");
53  asm.sendMessage(new AuthNAck(), " address ");
    }
55
    public static boolean authValid(AuthenticateUser signal,
        GWServerSM asm){
57      return false;
    }
59
}
```

Listing A.7: GWServer.java

A.6 User.java

```
package anonymousauthentication.user;

import anonymousauthentication.RegAck;
import anonymousauthentication.RegRequest;
import anonymousauthentication.AnonRegReq;
import anonymousauthentication.UserRegistered;
import anonymousauthentication.ReqCred;
import anonymousauthentication.Cred;
import anonymousauthentication.ReqTempCred;
import anonymousauthentication.InitiateAuth;
import anonymousauthentication.ReqAuthenticate;
import anonymousauthentication.AuthAck;
import anonymousauthentication.ReqTokenCred;
import anonymousauthentication.TokenCred;


public class UserActions extends Object {

  public static void sendRegRequest(RegAck signal, UserSM asm){
  asm.sendMessage(new RegRequest(), " address ");
  }

  public static void U_KeyToShares(RegAck signal, UserSM asm){
      // Code for invoking the Python code for creating shares
          of U_Key
  asm.sendMessage(new AnonRegReq(), " address ");
  }

  public static void sendReqCred(UserRegistered signal, UserSM
      asm){
  asm.sendMessage(new ReqCred(), " address ");
  }

  public static void sendRegRequestTest(UserSM asm){
  asm.sendMessage(new RegRequest(), " address ");
  }

  public static void sendReqTempCred(Cred signal, UserSM asm){
  asm.sendMessage(new ReqTempCred(), " address ");
  }

  public static void sendReqAuthenticate(InitiateAuth signal,
      UserSM asm){
  asm.sendMessage(new ReqAuthenticate(), " address ");
  }
```

```
44    public static void sendReqTokenCred(AuthAck signal, UserSM asm
          ){
      asm.sendMessage(new ReqTokenCred(), " address ");
46    }

48    public static void AccessResources(TokenCred signal, UserSM
          asm){
      }
50
  }
```

Listing A.8: User.java

Bibliography

[1] Oded Goldreich. *Secure Multi-Party Computation*. Department of Computer Science and Applied Mathematics Weizmann Institute of Science, Rehovot. October 2002. http://www.wisdom.weizmann.ac.il/õded/pp.html.

[2] E. Hammer-Lahav. *The OAuth 1.0 Protocol*. IETF RFC 5849, ISSN: 2070-1721, 2010.

[3] Andrew C. Yao. *Protocols for secure computations*. Foundations of Computer Science, 1982. SFCS 08. 23rd Annual Symposium, pages 160-164, 1982.

[4] Pedersen, Torben. *Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing*. Advances in Cryptology CRYPTO 91, 1992, pages : 129-140.

[5] T. Rabin and M. Ben-Or. *Verifiable secret sharing and multiparty protocols with honest majority*. STOC '89 Proceedings of the twenty-first annual ACM symposium on Theory of computing, ISBN:0-89791-307-8

[6] Håvard Vegge. *Master Thesis: Realizing Secure Multiparty Computation*. Department of Telematics, Norwegian University of Science and Technology, Norway, June 2009.

[7] Michael Ben-Or, Sha. Goldwasser, and Avi Wigderson. *Completeness theorems for non-cryptographic fault-tolerant distributed computation*. STOC 88: ACM Symposium on Theory of Computing, New York, NY, USA, 1988, pages 110.

[8] David Chaum, Claude Crépeau, and Ivan Damgård. *Multiparty unconditionally secure protocols*. STOC 88: ACM Symposium on Theory of Computing, New York, NY, USA, 1988, pages 11-19.

[9] O. Goldreich, S. Micali, and A. Wigderson. *How to play any mental game*. STOC 87: ACM Symposium on Theory of Computing, New York, NY, USA, 1987, pages 218-229.

[10] Y. Frankel, P. Gemmell, P. MacKenzie, and M. Yung. *Proactive RSA*. CRYPTO'97: International Cryptology Conference on Advances in Cryptology, 1997, , pages 440-454.

[11] A. Shamir. *How to Share a Secret*. Communications of the ACM, 22(11):612-613, 1979.

[12] Wade Trappe and Lawrence Washington. *Introduction to Cryptography with Coding Theory*. Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2006.

[13] Atle Mauland. *Master Thesis: Realizing Distributed RSA using Secure Multiparty Computations*. Department of Telematics, Norwegian University of Science and Technology, Norway, July 2009.

[14] Branden Archer and Eric W. Weisstein. *Lagrange Interpolating Polynomial*. MathWorld-A Wolfram Web Resource, http://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html, Last Accessed: June 2011.

[15] Official Webpage: Virtual Ideal Functionality Framework, http://viff.dk/doc/index.html, Last Accessed: June 2011.

[16] G.R. Blakley. *Safeguarding cryptographic keys*. National Computer Conference, 48:313-317, 1979.

[17] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. *Multiparty computation goes live*. Cryptology ePrint Archive, Report 2008/068, 2008.

[18] Ivan Damgård and Marcel Keller. *Secure Multiparty AES*. Financial Cryptography'10, 2010, pages: 367-374.

[19] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. *Fairplay - a secure two-party computation system*. USENIX Security Symposium, pages 287-302, 2004.

[20] Assaf Ben-David, Noam Nisan, and Benny Pinkas. *Fairplaymp: A system for secure multi-party computation*. CCS 08: ACM conference on Computer and communications security, pages 257-266, New York, NY, USA, 2008. ACM.

[21] Dan Bogdanov, Sven Laur, and Jan Willemson. *Sharemind: A framework for fast privacy-preserving computations*. Cryptology ePrint Archive, Report 2008/289, 2008.

[22] Simap - secure information management and processing, The Alexandra Institute Centre for IT security, Last Accessed: June 2011, http://www.alexandra.dk/uk/Projects/Pages/SIMAP.aspx.

[23] Martin Geisler, Ivan Damgård, and Benny Pinkas. *MPC virtual machine specification*. Technical report: Computer Aided Cryptography Engineering, January 9, 2009. http://www.cace-project.eu/downloads/deliverables-y1/CACE_D4.3_MPC_Virtual_Machine_Specification.pdf.

[24] E. Hammer-Lahav, Ed., D. Recordon and D. Hardt. *The OAuth 2.0 Authorization Protocol*. Internet draft, IETF, draft-ietf-oauth-v2-16, May 2011.

[25] Castejón, Humberto Nicolás and Bræk, Rolv. *A collaboration-based approach to service specification and detection of implied scenarios*. SCESM '06: International Workshop on Scenarios and state machines: models, algorithms, and tools, 2006, isbn: 1-59593-394-8, Shanghai, China, pages: 37–43.

[26] Bræk, Rolv., Oystein Haugen, and Ystein Haugen. *Engineering Real Time Systems: An Object-Oriented Methodology Using SDL (The BCS Practitioner)*. Prentice Hall, 1993. 416 pages, ISBN–10: 0130344486, ISBN–13: 978-0130344489.

[27] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. *Theory of Cryptography*, 2005, pages 342–362.

[28] Twisted Documentation. Webpage http://twistedmatrix.com/documents/current/core/howto/index.html. Last accessed: June 2011.