Die approbierte Originalversion dieser Diplom-/ Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

TU WIEN FAKULTÄT FÜR !NFORMATIK Faculty of Informatics

A Public Resource Computing Application based on the Secure Peer Model

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Matthias Lettmayer, BSc
Matrikelnummer 00728179

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. eva Kühn
Mitwirkung: Projektass. Dipl.-Ing. Stefan Craß

Wien, 5. März 2018

Matthias Lettmayer   eva Kühn

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Public Resource Computing Application based on the Secure Peer Model

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Matthias Lettmayer, BSc
Registration Number 00728179

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. eva Kühn
Assistance: Projektass. Dipl.-Ing. Stefan Craß

Vienna, 5th March, 2018

Matthias Lettmayer   eva Kühn

Erklärung zur Verfassung der Arbeit

Matthias Lettmayer, BSc
Sindelargasse 38, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. März 2018

Matthias Lettmayer

Acknowledgements

I like to thank my professor eva Kühn and her research assistant Stefan Craß for providing such an interesting topic. Furthermore, I would like to thank Stefan Craß for supporting me by insightful discussions and giving me constructive feedback.

Special thanks go to my wife and family, who were always there for me when I needed them. Without their help this thesis would have never been finished.

Kurzfassung

Das Peer Modell, entwickelt von der Space-Based-Computing-Gruppe an der TU Wien, führt eine Modellierungssprache ein, welche das Entwerfen und Implementieren von verteilten Systemen durch Verwendung von vordefinierten und anwendungsspezifischen Komponenten erlaubt. Üblicherweise müssen verschiedene Stakeholder in solchen verteilten Umgebungen Daten untereinander austauschen. Da eine Organisation in einem unsicheren Netzwerk, z. B. dem Internet, ihren Kommunikationspartnern nicht vertrauen kann, wurde das Peer Modell kürzlich um ein Sicherheitsmodell erweitert. Dieses enthält einen individuell anpassbaren rollenbasierten Autorisierungsmechanismus und eine attributbasierte Darstellung von Identitäten, welche eine hierarchische Delegation unterstützt. Des Weiteren wurde eine Sicherheitslaufzeitumgebung, namens *Secure Peer Space*, eingeführt. Da diese Sicherheitserweiterung relativ neu ist, existieren dafür noch keine Vorzeigeprojekte.

Diese Diplomarbeit beschäftigt sich mit dem Design und der Implementierung einer Public-Resource-Applikation, welche die gemeinsame Nutzung von Computerressourcen über ein Peer-to-Peer-Netzwerk ermöglicht und dadurch ein Beispiel für eine Secure-Peer-Space-Anwendung darstellt. Benutzer können eigene Projekte anlegen oder an fremden teilnehmen, um ihre CPU-Leistung mit anderen zu teilen. Projekteigentümer sind in der Lage, Teilnehmer durch generische attributbasierte Zugriffsregelungen auszuschließen. Die Autorisierung wird durch den *Secure Peer Space* überprüft. Arbeitspakete werden in *Tasks* gekapselt und an die Teilnehmer eines Projektes als *Subtasks* verteilt. Da der *Secure Peer Space* keine Authentifizierungsmethode besitzt, welche in einem realistischen Szenario zum Einsatz kommen kann, ist es auch Teil dieser Arbeit, eine zertifikatbasierte Authentifizierungskomponente für den *Secure Peer Space* zu entwerfen und entwickeln. Aus diesem Grund werden Benutzer durch X.509-Zertifikate, welche von einer selbstständigen Zertifizierungsstelle signiert werden, authentifiziert und die gesamte Kommunikation mittels TLS verschlüsselt. Das Peer Model wird verwendet, um die Koordinationslogik der Anwendung zu modellieren.

Die Umsetzung der Applikation wird durch eine Evaluierung bewertet, welche durch ein praktisches Beispiel die Erfüllung der Anforderungen nachweist und mit einem Benchmark-Test die Performance der Anwendung und des zugrunde liegenden Secure Peer Space analysiert.

Abstract

The Peer Model, developed by the Space-Based Computing group at TU Wien, introduces a modelling language that allows designing and implementing distributed systems by using predefined and application-specific components. Typically, different stakeholders need to exchange data in such distributed environments. As an organisation can not trust its communication partners in an insecure network, e.g. the Internet, the Peer Model was recently extended by a security model that enables a fine-grained rule-based authorisation mechanism and attribute-based representation of identities with support of chained delegation. Furthermore, a secure runtime architecture called Secure Peer Space was introduced. As this security extension is relatively new, no showcases exist yet.

This thesis provides such an example by describing the design and implementation of an enhanced public resource application that allows sharing computational resources over a peer-to-peer network. Users can create own projects or participate in them to share CPU power with each other. Project owners are able to exclude users by using generic attribute access rules. The authorisation is handled by the Secure Peer Space. Workload is encapsulated into tasks and distributed as subtasks to the participants of a project. As the Secure Peer Space lacks an authentication method that can be used in real world scenarios, it is also part of this thesis to design and implement a certificate-based authentication component for the Secure Peer Space. Therefore, users are authenticated by X.509 certificates, that are signed by an autonomous certificate authority, whereas the communication is encrypted using TLS. The Peer Model is used to model and implement the coordination logic of the application.

The implementation of the application is evaluated by a practical example proving the fulfilment of the requirements and a benchmark test analysing the performance of the application including the underlying Secure Peer Space.

Contents

Kurzfassung **ix**

Abstract **xi**

1 Introduction **1**
1.1 Aim of Work 2
1.2 Methodology 2
1.3 Structure of this Thesis 3

2 Related Work **5**
2.1 Public Resource Computing (PRC) 6
2.1.1 distributed.net 6
2.1.2 BOINC 6
2.1.3 Mining@Home 8
2.1.4 XtremWeb 9
2.1.5 Folding@Home 10
2.2 Comparison 10

3 Background **13**
3.1 Peer Model 13
3.1.1 Peers 13
3.1.2 Wirings 14
3.1.3 Secure Peer Model 14
3.2 Public Key Infrastructure (PKI) 17
3.2.1 Cryptography 18
Secret Key Encryption 18
Public Key Encryption 18
3.2.2 Digital Signature 19
3.2.3 X.509 Certificates 19
3.2.4 Public Key Certificate (PKC) 20
3.2.5 Attribute Certificate (AC) 21
3.2.6 Certificate Service Provider (CSP) 22
3.2.7 Certificate Revocation 23

3.2.8 Transport Layer Security (TLS) . . . 24
3.3 Programming Tools . . . 24

4 Requirements **27**
4.1 Functional Requirements . . . 27
4.2 Non-Functional Requirements . . . 28
4.3 Attack Scenarios . . . 29
4.3.1 Man in the Middle . . . 29
4.3.2 Impersonate other user/company . . . 29
4.3.3 Deliver wrong results . . . 29
4.3.4 Malicious code . . . 30
4.3.5 Altering points . . . 30

5 Design **31**
5.1 Architecture . . . 31
5.2 Certificate-based Authentication Component . . . 34
5.3 Functional Features . . . 35
5.3.1 User Registration . . . 35
5.3.2 Project Registration . . . 36
5.3.3 Task Distribution . . . 37
5.3.4 Point Submission . . . 38
5.4 Peer Model . . . 39
5.4.1 UserPeer . . . 39
Security Rules . . . 41
5.4.2 UserCAPeer . . . 41
Security Rules . . . 41
5.4.3 ProjectPeer . . . 42
Security Rules . . . 43
5.4.4 TaskPeer . . . 45
Security Rules . . . 45
5.4.5 WorkPeer . . . 46
Security Rules . . . 47
5.4.6 PointsPeer . . . 47
Security Rules . . . 48

6 Implementation **51**
6.1 Overview . . . 51
6.2 Tasks . . . 51
6.3 Adaptations made on the Secure Peer Space . . . 54
6.3.1 Add and remove discovery listeners . . . 56
6.3.2 Send to multiple destinations . . . 56
6.3.3 Send with timeout . . . 56
6.3.4 Certificate-based authentication . . . 56
6.3.5 TLS communication . . . 57

6.3.6 Modifications on configuration file 57
6.4 Interaction with the Secure Peer Space 58
6.5 Application Usage 61

7 Evaluation **69**
7.1 Fulfilled Requirements 69
7.1.1 Protections against Attack Scenarios 70
7.2 Use Case 71
7.2.1 Setup 71
7.2.2 Process 72
7.2.3 Fulfilled Requirements 74
7.3 Benchmark Test 74
7.3.1 Interpretation of Result 75
7.4 Comparison to the Analysed Systems from the Related Work 76
7.5 Critical Reflection 77

8 Future Work **79**

9 Conclusion **81**

List of Figures **83**

List of Tables **85**

Listings **87**

Acronyms **89**

Bibliography **93**

CHAPTER 1

Introduction

Public Resource Computing (PRC) [And04, BEJ+09, Rie12] uses the resources of personal computers to do computing-intensive (mostly scientific) tasks, such as finding extraterrestrial life (SETI@Home [Kor16]), generating an accurate three-dimensional model of the Milky Way galaxy (MilkyWay@Home [Des16]) or helping to heal global diseases like AIDS, cancer or Alzheimer (Folding@Home [Pan13], GPUGrid [dF16], etc.). Before PRC came up in the mid-1990s, scientific organizations used supercomputers or grid networks for their work, which are very expensive and optimized for only one specific type of task. On the other hand in PRC the scientists have no control over the computers of the participants. A user could for example deliver wrong results to acquire higher reputation by doing less work or a project owner could embed malicious code into a project. Moreover, security plays an essential role in such environments and PRC systems need to have techniques which prevent such attacks, so that scientists and participants can trust each other.

The Peer Model [KCJ+13], developed at TU Wien, is a modelling language for distributed systems, which perfectly fits to PRC applications. It consists of peers acting as individual components of a distributed system, which use a concurrent communication pattern. Each peer has a peer-in-container (PIC) and a peer-out-container (POC), which can hold entries of data. Wirings allow executing application-specific logic, generating new entries, moving entries between PIC and POC, and communicating with other peers via the destination (DEST) property. They are triggered depending on specified conditions (guards), which consist of queries over available entries. The Java implementation of the Peer Model is called *Peer Space*. It runs declared wirings and establishes remote communication between peers. Recently the Peer Space was extended by a security concept [CJK15], which makes use of attribute-based access control to allow authorized access on peer containers. In addition, entries that are sent to other runtimes are going through an authentication process. The Secure Peer Space is relatively new and therefore no showcases exist.

1.1 Aim of Work

The main aim of this thesis is the development of an enhanced public resource computing application with a flexible coordination system and extended security features, which should be implemented with the help of the Secure Peer Space. The program should allow users to define their own projects for computing-intensive task and to distribute them to other users, which can subscribe to projects. The project owners can specify which users are allowed or not allowed to participate on their projects by using generic rules. Participants of projects receive subtasks from the project owner and return the calculated results back to the owner. Incoming messages of other users should be authenticated by using certificates signed by an own central certification authority. Participants are allowed to pause a project, so that they have control over their CPU load and do not get any new tasks from the project owner.

The communication and coordination between the users should be designed by using the Peer Model. Afterwards the Secure Peer Space should be used to implement the created models in Java. As the Secure Peer Space supports only simple password authentication yet, an integration of a certificate-based authentication component is also part of this work.

At the end of the thesis an evaluation of the application should show that the Peer Model offers enough expressiveness to model the desired coordination processes and security features of the program and that the Secure Peer Space middleware works as expected regarding security, performance and usability.

The thesis should clarify if the Peer Model and its security model are suitable for realising a PRC application. Furthermore, it should be highlighted which novel features in the field of PRC are enabled by the middleware. The challenges of the work were the development of a complex distributed application by using a new programming model with security extension. Moreover, the development and integration of a secure authentication method that is usable for real world scenarios is not a trivial task.

1.2 Methodology

The methodological process consists of the following steps:

Literature review The first step includes the search for related work and gathering of background knowledge about distributed and public resource computing.

Documentation and design The requirements should be defined and the architecture and GUI should be designed.

Implementing authentication The Secure Peer Space should be extended with a certificate-based authentication component.

Implementing application The public resource computing application should be implemented by using the Secure Peer Space.

Evaluation The last step should show how secure and efficient the application is. For this purpose software tests and benchmarks should be used.

1.3 Structure of this Thesis

The next chapter gives insights into similar systems and compares them. Chapter 3 gives background information to the reader. It explains the Peer Model and Secure Peer Space. Afterwards, basic knowledge about Public Key Infrastructure (PKI) is provided. A list of used programming tools concludes this chapter. Functional and non-functional requirements for the application are shown in Chapter 4. Chapter 5 presents the design of the system by illustrating an architectural overview as well as explaining applied mechanisms. Additionally, important functional features as well as the authentication process are described. At the end of the design chapter, a detailed summary of the constructed Peer Model, including peers and security rules, is given. Chapter 6 shows the implementation of the PRC application. A class diagram depicts an overview of the implemented system, followed by an illustration of realised task types. A list of adaptations made on the Secure Peer Space is given, before an explanation of the user interfaces completes the implementation part. The evaluation is done in Chapter 7. An example for the application shows the practical use and proves the correctness of the implementation. The performance is analysed by a benchmark test. At the end a comparison to the analysed systems from the related work is depicted. Chapter 8 describes tasks that can be done in the future. Chapter 9 briefly summarises the thesis and gives a conclusion about the whole work.

CHAPTER 2

Related Work

Peer-to-peer (P2P) systems [MKL+02] allow users (peers) to communicate with each other in a decentralised way, i.e. a peer can be a client and a server at the same time. Peers work autonomously as no centralised server exists and are able to join or leave the network at any time. Moreover, P2P networks are highly scalable, because no centralised bottleneck exists. Another property of P2P systems is that the maintenance costs are shared between all participants, in contrast to a centralised system where the server bears the majority of the expenses. As peers communicate directly with each other, trust and security plays an essential role in P2P systems. For example, sandboxing techniques protect client machines from malicious code (XtremWeb [CDF+05]), reputation systems help to isolate misbehaving peers (eMule [KB05]) or certificates are used to authenticate peers in the network (PAST [DR01]).

A sub-domain of P2P is crowdsourcing [Bra08], which is defined as the process of outsourcing a job to a large group of people. For this an organisation creates a request that is presented on an Internet platform. The task can be solved by an individual person or collaboratively by dividing the work for an aggregated result. The participants receive rewards for solving the jobs.

PRC makes use P2P techniques and is very similar to crowdsourcing as volunteers help organisations to solve big problems. The difference is that the jobs are not solved by humans but by provided resources from their private computers (e.g. CPU or storage).

This chapter inspects existing PRC systems that are similar to the application this thesis is about. Subsequently, some projects are presented and compared concerning architecture, authentication, and security.

2.1 Public Resource Computing (PRC)

PRC, also known as Volunteer Computing, is a global network consisting of volunteers, who provide their unused computational resources (such as CPU/GPU cycles) to an organisation for solving very computing-intensive tasks. The first important projects were Great Internet Mersenne Prime Search [WK17] (1996), a system that collectively searches for large prime numbers, and distributed.net (see Section 2.1.1). In the following PRC, applications that relate to the work in this thesis are introduced and explained in detail.

2.1.1 distributed.net

Distributed.net [Dis16] is a PRC platform, which was founded in 1997 and is owned by the non-profit organisation Distributed Computing Technologies Inc. (DCTI). It uses idle CPU and GPU times of volunteers to solve distributable problems, including encryption challenges from RSA Data Security. Currently they are working on the RC5-72 project, which tries to decrypt a 72-bit encoded RSA message. The user which finds the right key first will receive a donation of 1 000 US Dollars.

The architecture consists of keyservers and clients lined up as a pyramid. At the bottom clients request data and send results to proxy keyservers. The master keyserver lies at the top and manages input data by sending unprocessed data and receiving results to/from intermediate proxy servers.

Delivered results from the clients can be easily checked for correctness by trying to solve the challenge. Therefore, no complex result fake prevention mechanisms are needed. A statistics website gives information about rankings, total checked keys, percentage of the exhausted keyspace and how much time was spent. There exist separated views for individuals and teams.

2.1.2 BOINC

Berkeley Open Infrastructure for Network Computing (BOINC) [And16a] is an open-source project which was released in the year 2002 by the work of many scientists, students and volunteers all over the world. It is a platform for public resource and volunteer computing. This means that it gives private people the opportunity to spend their computational power for specific scientific projects on a voluntary basis.

BOINC offers a system to register to different projects, which are created and hosted by institutions and organizations all over the world. There exists an Application Programming Interface (API), which allows scientists and software developers to implement their own scientific application and integrate it into the BOINC system [Rie12, And04]. As at July 2017 over 30 projects are registered at the platform [And16a].

Some examples are:

SETI@Home [Kor16, ACK+02] searches for extraterrestrial life by analysing radio telescope data from the Arecibo radio observatory in USA.

GPUGrid [dF16] uses the Graphics Processing Unit (GPU) power of project members to perform biomolecular simulations for getting a better understanding of diseases like cancer, AIDS and neural disorder.

Einstein@Home [All16] tries to prove that gravitational waves exist and searches for pulsars, fast spinning neutron stars. The project uses data from the Laser Interferometry Gravitational Observatory (LIGO), the Arecibo radio telescope, and the Fermi gamma-ray satellite.

Milkyway@Home [Des16] uses data from the Sloan Digital Sky Survey to generate an accurate three-dimensional model of the Milky Way galaxy.

For a project registration the user has to provide an email address and a password. Both are used to generate a key, which is used to authenticate the user.

There are only legitimated projects in the project list, which are known to be secure to BOINC. If users join a project which is not in the list by using an own project Uniform Resource Locator (URL), they have to make sure it does not have a malicious intent. The executables of a project are transferred to the clients using a secure communication channel (Hypertext Transfer Protocol Secure (HTTPS)). The code is signed using certificates and only programs with valid signatures (verified with the certificate of the project server) will be executed by the BOINC client [And16b]. When researchers create new projects at BOINC, they have to generate a key pair first. The project server must be provided by the researcher and must contain the public key certificate that is used by clients to verify signed executables. The private key, used to create the signatures, should be kept on a machine which is physically secured and isolated from the network [And07].

BOINC gives credits (points) to the user for successfully and correctly delivered workunits. The calculation of the points is project-specific but normally depends on the CPU time spent for the task. To ensure that the results are not faked, the same workunit will be processed by multiple clients. Project owners can choose how multiple results of a unit must look like to be accepted as a correct outcome by the system. For example if all results of a workunit are identical or in a specified range, the users will get the points for it, otherwise the task needs to be computed by other clients to get further results [Rie12].

The architecture of BOINC is modularly designed and consists of the following components:

Database stores informations of applications, platforms, versions, workunits, results, user accounts, etc.

Scheduler sends workunits to clients and processes completed results.

Data server holds the application-specific parameters and provides functions to upload (secured by a certificate-based mechanism) and download (plain HTTP) files.

Client is installed on the volunteer's machine and executes workunits of registered projects received from the data server. Completed results are transferred to the scheduler.

Transitioner handles state transitions of workunits and results as well as identifies error conditions.

Validater controls the results of same workunits and selects the correct one by using an application-specific comparison function.

Assimilater inserts the correct result into the database.

File deleter removes files from the data server that are not needed any more.

All components are decoupled from each other and can be run on different servers. Moreover, it is not required that all parts of the system are always available, e.g. the scheduler and transitioner can still work if the database is down. Client connections can be scaled to several servers placed at different locations [And04].

2.1.3 Mining@Home

Mining@Home [LMOT10] combines data mining applications with public resource computing. It uses a super-peer network [MTV05] with caching techniques to solve the Closed Frequent Itemsets Mining problem (CFIM), where frequently occurring data needs to be found in a huge amount of data sets.

The framework makes use of the following components:

Job Manager creates job adverts and collects the output results.

Miners execute the jobs and are connected to (exactly) one super-peer.

Super-Peer acts as the backbone of a network of miners (grid) and is connected to other super-peers through a high-level P2P network.

Data-Source stores the data inputs of the jobs.

Data-Catcher retrieves data from data source or other catchers and saves it temporarily.

A miner sends a job query to the nearest super-peer, which forwards the message to the job manager if the address is known. Otherwise, the job query can travel through many super-peers until it arrives at the job manager. The manager returns a job advert, which matches the job query, to the miner. A job advert specifies the task-specific parameters and the input data. To retrieve the input data of a job, the miner issues a data query to

retrieve the location of a data-catcher, which downloads the resources from a data source or another catcher. The data-catcher passes the input data to the requesting miner and stores it also for other miners.

If a super-peer detects that one of its miners is terminated and has not finished a job, it returns the corresponding job advert to the job manager, so that the job can be reassigned to another miner. This process is only possible because of the decentralized architecture of the super-peer network, which allows adding and remove miner units dynamically. Furthermore, miners only need to communicate with their neighbour (nearest) super-peer and do not need to know any further network addresses.

2.1.4 XtremWeb

XtremWeb [CDF+05] is a public resource computing middleware for large-scale distributed systems written in Java. It uses a modularly designed P2P architecture with three tiers: client, coordinator and workers.

The worker consists of four components: task pool (saves state of tasks), execution threads (does computation), communication manager (downloads/uploads files) and activity monitor (checks if CPU is idle). The client can submit task requests and retrieve results from completed jobs. The coordinator acts as a mediator by decoupling clients from workers. It takes job requests from clients and assigns them to available workers. On the other side the coordinator retrieves the results from workers and forwards them to corresponding clients. The parties must exchange coordination messages and input data, which is needed for executing the tasks. For this Remote Processor Call (RPC) is used, where bindings for diverse implementations exist (e.g. JavaRMI, XML-RPC, JXTA). Furthermore, the communication between the entities is established by using Secure Sockets Layer (SSL), which encrypts the messages and assures the authenticity of the parties.

Clients, coordinators and workers are independent entities that can be run on different machines. Furthermore, they can be stopped and restarted at any time without the system to fail. The coordinator recognizes halted worker units and tries to re-assign the uncompleted task to other workers.

To protect the workers from malicious code, XtremWeb makes use of sandboxing techniques using SBLSM, a module for Linux Security Modules (LSM) [MSKH02] tailored to P2P systems. A system call can be granted or denied, or an authority can be asked what to do. Three different control levels exist:

1. File access control allows to restrict the access to specified files for sandboxed processes.
2. Network access control uses black- and whitelists to control network connections.
3. Process signal control allows the restriction of signal messages the sandboxed process is allowed to send.

2.1.5 Folding@Home

Folding@Home [BEJ+09] is a public resource computing system released in 2000. Its main purpose is the simulation of protein folding to get a better understanding of diseases like Alzheimer, Parkinson and Bovine spongiform encephalopathy (BSE), where protein misfolding plays an essential role. The calculations are done on computers of volunteers, where each volunteer works on an own folding task.

Volunteers have to install a client software, which communicates with different Folding@Home servers. The work server creates new work units depending on previously received results. Each work unit points to a location of executable code (core), which is cryptographically signed and specific for the used platform (Windows, Linux, OS X, OpenBSD and FreeBSD). As the cores are exchangeable, the client software need not be updated or reinstalled if the code changes. A work server can host many projects, administrated by different scientists.

The assignment server schedules the work units across the clients. For this many factors are considered, e.g. time-critical results and client-side parameters. For each completed work unit the user receives points (credits), which are saved to the assigned account and team. A statistical website allows the volunteers to see how much they have contributed and how much credits they have earned for their team.

2.2 Comparison

A comparison of important PRC criteria of the analysed systems is shown in Table 2.1.

In Peer-to-Peer networks the endpoints can communicate directly with each other without needing any permanently running server. Scalability allows an increase of resources (e.g. users) without any change in the architecture or source code. Some PRC applications are able to define and execute custom tasks. An access control verifies if a user is authorised to perform a desired action. A secure communication channel allows partners to talk secretly with each other and prevents eavesdropping and man-in-the-middle attacks. Secure code execution hinders running of malicious executables on client machines. A reward system tries to motivate users to share their computational resources by giving them in return some compensation (e.g. points/credits, reputation or money).

Although distributed.net, BOINC and Folding@Home rely on a client-server architecture rather than using a Peer-to-Peer network, they are rated with $\sim$ in scalability. They have many servers distributed all over the world and make use of load balancing techniques. Therefore, these applications are able to intercept thousands of client requests per second and redirect them to servers with low workload. Nevertheless, the scalability of such systems highly depends on the server availability and the load balancer, which both will cost a fair amount of money for managing such high request numbers are vulnerable to distributed denial-of-service (DDOS) attacks.

	distr.net	BOINC	M@Home	XWeb	F@Home
P2P	−	−	+	+	−
Scalability	∼	∼	+	+	∼
Custom tasks	−	+	−	+	−
Access control for PO	−	−	−	−	−
Secure communication channel	−	+	−	+	−
Secure code execution	−	+	−	+	+
Reward system	+	+	−	−	+

Table 2.1: Comparison of relevant features from analysed related systems

+: supported
∼: supported with limitations
−: not supported
PO: Project Owner

In BOINC it is possible to define custom projects by using its C++ API. Because XtremWeb acts only as a middleware, it can be tailored to meet any task it is required for.

XtremWeb uses access control to protect users from malicious code, but none of the analysed systems uses an access control model that lets project owners choose their participants.

BOINC communicates over HTTPS with the client machines to improve security. XtremWeb uses SSL to encrypt the communication between their tiers.

Code signing helps BOINC and Folding@Home to execute only trusted programs, whereas XtremWeb runs code in a secured environment (sandbox).

Users will receive credits for correctly completed tasks in BOINC and Folding@Home. If the user is part of a team, it will also receive the points. RSA Labs is going to give a 10 000 US-Dollar reward to the first one finding the solution to their RC5-72 secret key challenge. If a distributed.net machine will find the right key, the registered user and team will receive each 1 000 $, 2 000 $ will be kept by distributed.net and the rest will be donated to a non-profit organisation, selected by the community.

None of the investigated systems fulfils every attribute listed in the above table. Therefore, the motivation for the own application is to implement all criteria.

CHAPTER 3

Background

This chapter gives information about the Peer Model and Secure Peer Space as well as fundamental knowledge about PKI. This should help the reader to understand subsequent chapters of the thesis better. Concluding, the used programming tools are presented.

3.1 Peer Model

The Peer Model [KCJ+13] introduces a modelling language for coordinating data-driven workflows of distributed systems in a high-level view of abstraction. It separates the coordination from application logic, which increases readability and maintainability. Peers, which form the main units of the Peer Model, are loosely coupled reusable components. They interact with each other by sending/receiving asynchronous data (entries) stored in space containers [KMKS09] similar to Linda tuple spaces [Gel85]. In the following, the individual components of the Peer Model are described.

3.1.1 Peers

A peer consists of a Uniform Resource Identifier (URI), a PIC for receiving data and a POC for sending data. Data, which is written into and read out of the containers, is modelled as entries with additional meta-information divided into coordination- and application-specific data parts. Entries must have a specific type, which helps to model a workflow in the application. Further coordination properties are time-to-start (TTS) (entries which are not started yet are ignored), time-to-live (TTL) (expired entries are also ignored), flow ID (same IDs indicate that entries belong to the same workflow) and destination (DEST) (where should the entry be written to). Additional to writing an entry into a container it can also be read or taken (read and delete) from it.

While the PIC and POC are used to the save the state of a peer, the behaviour is modelled by using wirings (services) and sub-peers, i.e. nested peers. To run the peers

on a local machine a Runtime Peer is needed. It bootstraps the environment by enabling the dynamic creation and removal of peers.

3.1.2 Wirings

A wiring consist of guards, services and actions. One or more guards describe the entries which will trigger the execution of the wiring by means of link operations that transfer matching entries to a temporary internal space container, the entry collection (EC). A link operation is composed of a query, including a count parameter, and a type. The query defines an entry type and optionally filters further coordination properties by specified values (e.g. id = 10). A minimum and maximum number is used by the count parameter to describe a range of how many matching entries the guard should select. The link type can either be move (default), which removes the entries from the source container, or copy, which reads the entries from the source container. The selected entries are transferred to the EC. If the developer has defined a service method and all guards were fulfilled, then the service is called using the entries of the EC as input parameters. Services are the only way to insert application-specific logic into the Peer Model as they allow executing user-defined code. The result of the service method is placed back into the EC. Optional actions take specified entries, described by a link operation, from the EC and transmit them to their defined destination (DEST property). In contrast to the guards, where non-matching links will prevent the wiring from further execution, the action links are non-blocking operations. This means if a link operation is not fulfilled it will be skipped and the next action will be processed if one exists. The final step is to delete all remaining entries, which were not handled by an action, from the EC.

3.1.3 Secure Peer Model

The Secure Peer Model [CJK15] is an extension for the Peer Model. It introduces security features, like fine-grained rule-based authorisation and attribute-based access control, which are founded on previous work of an access control model [CDJ+13]. A principal (user or system) is uniquely described by a user id and a domain (e.g. organisation name). Further attributes can be included and depend on the use case. Peer owners can define access control rules to allow other subjects to read and/or take entries from their peer containers. A subject can be represented by a single principal or a delegation chain consisting of several principals. An access control rule is formed by using the following fields:

subjects Subject templates are matched to subjects for which the rule is applied.

resources The resources field describes peer containers, e.g. PIC, which are targeted by the rule.

operations This field specifies an access type operation (**read**, **take** or **write**) that the subject is allowed to perform.

condition Conditions allow to specify that the peer must be in a specific state. For example, an entry of the type T in the peer's PIC must have a coordination property ID with the value 1.

scope The scope field can be used to filter for specific kind of entries (i.e. entry types or coordination properties).

The Secure Peer Model allows for modelling chained delegations of entries. For example $User2$ has received an entry from $User1$ and wants to forward this entry to $User3$. By using direct access mode the entry owner is set to $User2$, whereas performing indirect access on the entry the initial owner ($User1$) will be preserved and a delegation chain is built ($User2$ **for** $User1$). If a peer receives an incoming entry, it first authenticates the entry owner. This is done with the help of an authentication chain, which functions as the delegation chain but saves the path of authenticated principals and their authenticators. In the example above the authentication chain would be ($User1$ **@** $User2$ **@** $User3$), as $User2$ authenticates $User1$ before processing the incoming entry of $User1$ and the same happens when $User2$ delegates the entry to $User3$. This concept relies on an explicit trust relationship between cooperating principals. This means that $User3$ trusts $User2$ that $User1$ was correctly authenticated and is not going to authenticate $User1$ again. Each principal in the delegation chain is associated with an individual authentication chain to preserve the whole path of the authentication. The final data structure is called subject tree and consists of the delegation chain plus the separate authentication chains. Each entry contains an individual subject tree that represents the *entry owner*. The local runtime user sits at the root of the tree and the principals of the delegation chain reside at the leaves. The authentication chains are mapped as paths from leaves to the root, whereas the delegation chain is determined by a left-to-right tree traversal.

Subject templates are used in the subjects field of access control rules. They are defined by a tree consisting of a set of predicates on subject properties for each node. The tree is compared to the subject tree of a principal and matches only if all predicates are fulfilled. The predicates can be defined with the help of comparison operators, like equality, inequality, bigger than, and smaller than or using wildcards for single nodes ("*") and chains ("**"). If there exists a principal that matches to the predicates of the templates, the access rule will be applied. An example for a subject template would be "$[id = User1,\ domain = OrgA]$ **for** (***** **@** ****** **@** $[domain = OrgB,\ credits \geq 1000]$)". This means $User1$ from an organisation $OrgA$ acts on behalf of another peer, which was authenticated directly or indirectly ("**") by an organisation $OrgB$ that has at least 1000 *credits*.

By using *context variables* the Secure Peer Model provides the possibility to define access rules that depend on the context of the accessing subject. The variables can be used instead of any coordination property value in the subject, scope or condition field of a rule. An example is the $\$originator$ variable, which represents the original creator of an entry.

Besides the PIC and POC, three new meta-containers were introduced through the Secure Peer Model. The Wiring Specification Container (WSC), Peer Specification Container (PSC) and Security Policy Container (SPC) allow to dynamically add or remove wirings, sub-peers and access rules. As wirings and peers are defined by entries in the WSC and PSC, the owners of such meta-entries are called *wiring owners* and *peer owners*. With the help of the new meta-containers it is possible that *peer owners* can allow other principals to administrate their peers by giving other entities access rights on the respectively WSC, PSC and/or SPC.

When a wiring tries to take or read an entry from a container, an authorisation process checks that the *wiring owner* has the right, defined by access rules in the SPC of the corresponding peer, to do this. On the other hand, if a wiring wants to write an entry to a container, it is checked that the *entry owner* (direct access) or the subject "*wiring owner* **for** *entry owner*" (indirect access) is allowed to perform this action. *Peer owners* are always permitted to directly access the containers of their own peers, which should prevent the creation of inaccessible and uncontrollable peers.

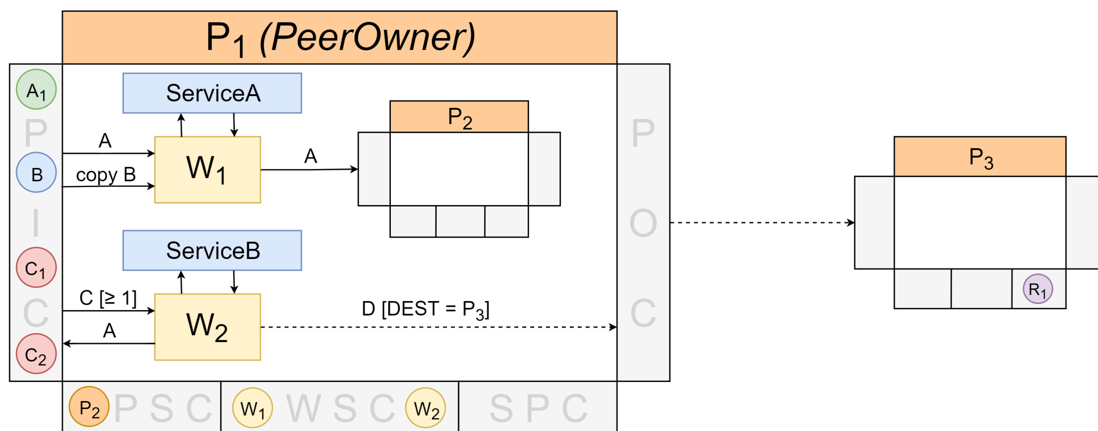


Figure 3.1: Example of a Secure Peer Model with peers, entries, wirings and sub-peer.

Figure 3.1 shows an example that makes use of the Secure Peer Model. Wiring W_1 consists of two guards and one action. The first guard takes an entry of type A, whereas the other reads (*copy*) an entry of type B from the PIC of peer P_1. The guards put the resulting entries, one A and one B, into an intermediate container, the EC. Then W_1 calls the function *ServiceA* and uses the entries from the EC as input parameters for the service. After the execution of *ServiceA*, its result will be transferred to the EC. The only wiring action of W_1 takes an entry of type A from the EC and transmits it to the PIC of sub-peer P_2. The wiring W_2 takes one or more entries of the type C from the PIC of P_1 and calls the *ServiceB* method using the extracted entries (Cs) as parameter. W_2 takes an entry A from the result of the service call and saves it into the PIC of P_1. Furthermore, an entry D is taken from the outcome and sent to another peer P_3. This operation is simplified by a dashed arrow, which symbolizes multiple wirings that

are created by the local runtime to forward the entry to its defined destination (DEST attribute).

The entries lying in the PIC (one A, one B and two Cs) represent the current internal state of P_1. Consequently, W_1 and W_2 can each be triggered one time. As W_2 outputs a new entry of type A, wiring W_1 is able to fulfil its guards condition one more time and can therefore be executed two times in total.

The meta-model is described by the entries located in the PSC, WSC and SPC. P_1 consists of a sub-peer P_2 and two wirings, W_1 and W_2. P_3 contains a rule R_1 that looks as follows:

NAME: R_1

SUBJECTS: $[ID = PeerOwner]$

RESOURCES: PIC

OPERATIONS: *write*

SCOPE: D

CONDITION: NOT $(PIC \triangleright D)$

This rule allows the peer P_1, which is owned by $PeerOwner$, to write entries of type D into the PIC of P_3. The condition permits this write operation only if there does not exist an entry of type D in the PIC yet.

The internal structures of sub-peer P_2 and peer P_3 are not modelled, due to the priority of creating a basic and not too complicated example.

3.2 Public Key Infrastructure (PKI)

By using public key cryptography users are able to encrypt data for communicating secretly or to sign data for ensuring the identity of the sender. Both operations make use of an asymmetric cryptosystem by using a key pair consisting of a public key for encrypting data and verifying signatures and a private key for decrypting data and generating signatures. A PKI binds an identity (e.g. unique name) with a public key by using digital certificates. Certificate authorities (CA) are used for issuing such certificates and take the responsibility that the provided authentication data of the issuer is correct. Furthermore, PKIs are responsible for managing key pairs through their whole life-cycle. Generating secure keys, publishing public keys, creating private key backups and handling expired and invalid keys are some tasks a PKI must be able to perform [Spi09, BKW13].

First a short description of encryption techniques and digital signatures is given. In the following X.509 certificates are explained. This includes a differentiation between public key certificates and attribute certificates. Subsequently, certificate service providers (e.g. CA) and methods to invalidate (revoke) certificates are illustrated. Finally, a short description of a PKI-based protocol, namely Transport Layer Security (TLS), is provided.

3.2.1 Cryptography

This section gives a short overview of cryptographic techniques, secret and public key encryptions, and should help to understand how encrypted communication works.

Secret Key Encryption

To ensure the confidentiality of data, secret key encryption uses the same key to encrypt and decrypt a message (symmetric cryptosystems). This secret key must be negotiated between the communication partners. One way of doing this is to use a secure channel, like a courier, but it is also possible to exchange the key with the Diffie-Hellman protocol [DH76]. Symmetric cryptographic techniques exist already since ancient times, e.g. Caesar cipher. In 1976 Data Encryption Standard (DES) [U.S99] was chosen as a Federal Information Processing Standard (FIPS) for the United States and was used worldwide from that time forward. Advanced Encryption Standard (AES) [U.S01] replaced DES in the year 2002 as a new standard. DES is not longer considered as a secure encryption technique, because it becomes possible to break the algorithm in a reasonable short time with specialized computer-hardware (e.g. COPACOBANA [KPP+06]) or super-computers (e.g. Deep Crack [Gil98]).

Public Key Encryption

Public Key Encryption uses two keys to establish a secret communication channel between two endpoints (asymmetric cryptosystems). Each communication partner is in possession of a public and a private key. The first one is used to encrypt messages and is accessible for everyone in the network, whereas the private key is capable of decrypting ciphertext, which was encrypted by the corresponding public key. It must be kept in a secret place, so that only the owner is able to read the encrypted messages. It is not possible to derive the private key from only knowing the public key in a reasonable amount of time. Furthermore, the security of asymmetric cryptosystems depends on the confidentiality of the private key and how the public key is tied to the identity, i.e. does the public key really belong to the person we want to send a message to. PKI uses certificates signed by a trustful authority to overcome this problem. To prevent the leaking of a private key, it can be put on a smartcard or encrypted with operating system facilities.

RSA [RSA78] was invented in 1977 by Rivest, Shamir and Adleman at the MIT. It was the first public key cryptosystems and is still widely used today.

The advantage of public key encryption is that for a confidential communication in an open network (e.g. Internet), it is not required to exchange a secret key like in symmetric cryptosystems. The sender has access to the published public key of the receiver and uses it to encrypt a message, which only the owner of the corresponding private key can read. Asymmetric cryptosystems are not as efficient as their counterpart. That is why a mix of both systems (Hybrid Encryption) is used in practice.

3.2.2 Digital Signature

This asymmetric authentication technique assures the authenticity and integrity of data. A private key is used for signing, whereas a public key allows performing a verification operation. If users want to show that a message was really sent from them and was not altered by someone else, they can use their private key to create a digital signature of the message. The receiver can verify the signature by using the public key of the sender. The verification procedure fails if the public key does not correspond to the private key used to generate the signature or when the message was modified.

3.2.3 X.509 Certificates

The X.509 model is a recommendation standard defined by the International Telecommunication Union Telecommunication Standardization Sector (ITU-T) [ITU18]. It is based on various Internet Engineering Task Force (IETF) [IET18] standards. Two types of certificates are declared in the X.509 standard: public key certificates that associate a public key to an entity (see Section 3.2.4) and attribute certificates, which assign privileges to entities (see Section 3.2.5).

Moreover, the X.509 is part of the X.500 directory standard [YHK93], which includes a naming structure called distinguished naming (DN). For creating a DN the following items can be used to define an entity.

common name (CN) This attribute is used to name the entity. For a person this would be the full name, whereas a web server should use the web address.

organisation (O), organisational unit (OU) The name and field of the organisation in which the entity works or is part of is specified by these fields.

country (C), locality/city (L), state/province (ST), street address (STREET) These attributes define the address at which the entity works or lives. Not all fields must be provided.

user identifier (UID) A unique identifier can be used to distinguish entities having the same DNs.

The abstract syntax notation version 1 (ASN.1) is a specification language that is used in X.509 certificates to describe the structure of data fields. Together with the distinguished encoding rules (DER) format the binary version of a certificate is created.

To allow the classification of attribute names and values, X.509 certificates use object identifiers (OIDs). They are build by a hierarchical name space, where each object is uniquely mapped to a positive number. The result is a tree, where internal nodes represent categories and leaves symbolize objects. For example, the hash function $SHA-256$ is written as the OID 2.16.840.1.101.3.4.2.1, or *common name* (meta-element) is assigned

to the ID 2.5.4.3. It is not necessary for a certificate to use OIDs, but it is recommended as it increases interoperability.

At the moment the X.509 standard comprises three versions of certificates. In 1988 the first version was enacted, which contains the following fields: version, serial number, issuer, subject, validity period, signature algorithm and value. The subsequent X.509v2 certificate was released in 1993 and extends the first version by two additional fields: issuer and subject unique ID. The last version, X.509v3, was specified in 1996 and allows the use of extensions.

Every certificate consists of the following fields:

version The version field contains a number specifying the version of the X.509 specification that is used in this certificate. The value is empty or 0 for v1, 1 for v2 and 2 for v3. At the present time X.509v3 certificates are the most used version.

serial number Each issuer is required to set this field to a non-negative number, which together with the issuer name uniquely identifies a certificate. The easiest way to achieve this is to assign a consecutive serial number to new certificates.

signature algorithm and value The signature algorithm contains the name of the algorithm (as OID), which the issuer used to sign the certificate (e.g. $SHA - 256$). The signature value contains a bit string that is generated by the issuer with the private key.

issuer The issuer is responsible that the certificate data is correct. By signing and issuing a certificate the issuer vouches for this. The name of this field is described by a DN.

validity period Two dates, *notBefore* and *notAfter*, define a time period in which the certificate is valid.

issuer unique ID To distinguish between issuers that have the same DN, a unique identifier can be written into this field.

extensions If a certificate needs to have additional attributes, they can be defined in this field. Every extension consists of an identifier (*extnID*), a boolean value (*critical*), which indicates if the extension must be evaluated, and a DER-encoded ASN.1 structure of the extension description (*extnValue*).

3.2.4 Public Key Certificate (PKC)

A public key certificate binds numeric public key to a human-readable subject and allows to send an encrypted message to the owner of the certificate.

In addition to the above listed attributes, a public key certificate (PKC) contains following extra fields:

subject The owner of the certificate is called subject and is in possession of the private key that corresponds to the public key embedded in the certificate. As for issuers, a DN represents the name of this field.

subject unique ID If different subjects have the same DN, this field could help to tell the difference between these two entities.

public key This field consists of an OID representing the name of the algorithm used to encrypt the public key, optional parameters used by the algorithm and the public key itself in ASN.1 format encoded as DER.

Figure 3.2 illustrates the fields of an X.509v3 public key certificate.

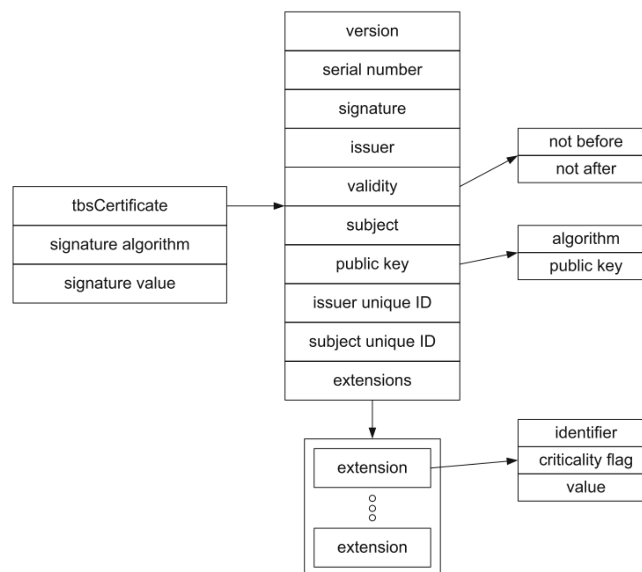


Figure 3.2: Structure of an X.509v3 public key certificate taken from [BKW13].

PKCs are typically used by web servers to provide an evidence of their issued identity. Furthermore, in the S/MIME protocol [Ram99] PKC can be used to encrypt and sign email messages. The lifetime of a PKC depends on the use case, e.g. server certificates have an average validity period of two years. X.509 certificates refer to public key certificates in literature, as they are more relevant than attribute certificates.

3.2.5 Attribute Certificate (AC)

In contrast to PKCs, attribute certificates (ACs) [De 11, Rex04] do not contain any public keys. Instead, they are filled with attributes, which allow binding privileges to the owner of the certificate and express what the holder is or is not allowed to do. Consequently, it can be used for distributed attribute-based access control (ABAC).

As ACs can only be used to authorise the owner of the certificate, it is necessary to use ACs together with an authentication system, e.g. PKCs.

ACs are specified in the ITU-T X.509 standard with the following additional fields:

holder This field is used to create a link to a PKC, whereby it specifies the owner of the AC. This can be done in three different ways: serial number, entity names and object digest information. The first one is the most common way as it simple uses the unique serial number of the PKC to connect the certificates. Entity names enable the possibility to use other authentication methods than PKCs. Object digest information allows the holder field to be a hash value calculated from an object representing the identity of the holder (e.g. public key or PKC) [Nyk00].

attributes The data placed into this field is application-dependent and is not restricted by any rules. For example a role or group attribute can be used for role-based access control.

The biggest advantage of separating authorisation attributes into an AC is the ability of applying a shorter lifetime to it than the PKC. This allows changing privileges of entities much faster (e.g. when a user becomes admin), while their identity stays unaffected and can be valid longer.

PKCs are issued by a CA, whereas ACs are certified by an own third party, namely an attribute authority (AA). It is possible that a CA and AA are the same entity, but normally they are split into separate institutions because of security reasons.

3.2.6 Certificate Service Provider (CSP)

A Certificate Service Provider (CSP) is in charge of managing the life-cycle of certificates. There exist three phases that a certificate goes through. The first one is the *generation phase*, where a new certificate is created and issued. This includes collecting and verifying information from the applicant, generating a key pair or gathering the public key from the requester. If everything went fine, the certificate is signed, delivered to the owner and published so that it can be used by other entities. In the *validity phase* the data of a certificate is checked for correctness. If validity period, signature, revocation status and certificate chain are all valid, the certificate information is reliable and can be used for further operations, like establishing a TLS connection with the certificate owner. Expired or revoked certificates pass over to the last stage of a certificate lifetime, the *invalidity phase*. It is best practice to archive invalid certificates, because they can be still useful (e.g to verify signatures in a certificate chain).

A CSP is divided into the following subcomponents:

registration authority (RA) If an entity requests a new certificate, registration is the first step that is necessary. This process is performed by the RA, which

collects information, like name, citizenship, place and date of birth, email address and biometric data, from the applicant. The gathered data does not only serve as certificate parameters, but also for contact and billing information as well as potential future legal disputes. The person who requests a certificate must come personally to the RA, provide the information and some proof of identity. Sometimes it is sufficient to submit the registration data online to the RA. Future certificate holders can decide if they deliver a self-generated public key or let the CSP do this job for them. At the end of the registration process, the RA verifies that the provided data is correct and that the applicant is authorized to receive the requested certificate. If this is successful, the RA sends a certificate signing request (CSR) to the CA.

certificate authority (CA) After receiving a CSR from the RA, the CA uses its private key to sign the request and issues a new certificate to the applicant. Other tasks of a CA are providing revocation information and key backup and recovery. The CA uses several security measurements to protect itself against outside attacks. For example, more than one entity is required for performing safety-related operations and it uses hardware security modules (HSMs) to keep its private key safe.

attribute authority (AA) An AA issues ACs and is the equivalent to a CA.

validation authority (VA) VAs are capable of revoking valid certificates and disburden the CA from this task.

directory service A directory service is helpful if the CA operates without a connection to the outer world. It is able to publish certificates and certificate revocation lists, in addition to transmitting certificates and private keys to the users.

3.2.7 Certificate Revocation

The procedure of invalidating a certificate during its lifetime is called revocation. There exist several reasons why a certificate must be revoked before its expiration time. For example, an attacker gets access to the private key that corresponds to the public key of the certificate.

Two established methods exist that allow revoking a certificate: certificate revocation list (CRL) and online certificate status protocol (OCSP). To invalidate a certificate, revocation information must be published in such a way that it is everywhere and always accessible. This data consists of the revoked certificate, a timestamp that indicates the point in time when the certificate was revoked and a reasons why it was invalidated. Moreover, it must be possible to prove the correctness of the revocation information by anyone.

CRL is a collection of revoked certificates that is signed by an issuer (CA or VA) and updated in regular periods of time. The whole CRL must be downloaded, to check if a certificate is still valid. Over time the list grows bigger and it will need more time to

retrieve the full CRL. For this reason *delta CRLs* were invented. They only contain the set of certificates that where revoked after the reporting of the last full CRL. Another problem is that CRL have a time period where a certificate is invalidated but not yet published as updates occur on a regular time basis (e.g. weekly on Monday). This means that a freshly invalidated certificate can only be seen as revoked on the next upcoming publication of the CRL.

In OCSP clients can ask a server that possesses up-to-date revocation information if a specific certificate is invalid. This allows obtaining nearly instantaneous revocation information after the certificate was invalidated. The OCSP server responds with the status of the requested certificate, which can be *good*, *revoked* or *unknown*. In contrast to CRL, such queries cost much lesser space. A downside of OCSP is that an Internet connection is required for the communication with the server, i.e. each request must be handled by the server, which obviously does not scale well. This is also the reason why this approach is not used by today's browsers.

3.2.8 Transport Layer Security (TLS)

TLS [DR08] is a PKI-based Internet protocol that is the successor of SSL [FKK11]. The protocol allows encrypted and authenticated communication between two entities. First a TLS Handshake protocol uses public key cryptography that authenticates the users and establishes a secure communication channel by generating unique symmetric keys for each session. This key is then used for the actual communication between the parties. TLS supports also client authentication, which allows the server to authenticate entities of incoming request. Furthermore, a secure hash function (e.g. SHA-256) can be used to assure the integrity of messages.

TLS is often used by web-based application that are operating in a security-relevant area (e.g. online-banking or e-commerce).

3.3 Programming Tools

In the following the applied programming tools are listed and described:

Java 8 [Jav17a] is a widely used programming language, which supports software development and program execution on multiple platforms (Windows, Linux, Mac, Solaris).

JavaFX [Jav17b] is an API that enables the functionality to create graphical user interfaces in Java. It replaces the old Swing widget toolkit and introduces new features, e.g. designing graphical user interfaces (GUIs) with a visual tool (Scene Builder), which transforms the layout into an FXML file, an XML-based markup language for JavaFX.

Eclipse [Ecl17] is an Integrated Development Environment (IDE) used mainly for Java, but there exist also plugins for other programming languages, like C/C++. Many extensions for the IDE are available, e.g. Git integration.

Apache Maven [Apa17] is a build and dependency system. It uses a project object model (POM) file to configure the build process and manages plugins and dependencies. Furthermore, it allows running test cases and creating documentation files, like Javadoc or a JUnit test report.

CHAPTER 4

Requirements

This chapter gives a short overview over the requirements of the application. At the end, possible attack scenarios with taken countermeasures are described. The main goal of this thesis is to develop a program that is able to share computational resources between users using the Secure Peer Space as basis for communication, coordination and storage. A simple user management should allow members to register and log in to the software. Afterwards users are able to create new projects, which build the foundation for using the CPU power of participants. Tasks belong to a certain project and describe a job that is divided and distributed to the contributors, who execute it and return the result to the project owner. An ABAC based on generic rules should regulate which users are allowed to register to a project. Each user is able to create projects and/or join projects of other users. As users are going to communicate directly with each other, security played an essential role in the elaboration of the requirements.

In the following functional and non-functional requirements are listed and explained in more detail.

4.1 Functional Requirements

Functional requirements describe the behaviour of the system. The inspiration of the features listed below came from other PRC systems that were analysed in the related work section.

Create projects A user can create projects by choosing from several project types. Through this action he/she becomes the owner of the project. The owner should be able to define rules, which allow or deny users from participating to the project (registration rules). These rules can also be generic, like block all usernames starting with letter 'a'.

Search for projects Users can search for projects and filter the results by defined criteria.

Project registration Users can sign up to a project if the project owner allows it through registration rules, and become participants of the project. Deregistration at any time should be possible.

Create tasks Project owners are allowed to create arbitrary tasks for their projects. A task has to include specific data, which is needed to execute it. Furthermore, the owner must specify in how many subtasks the task should be split and if a subtask should be executed on more than one machine (redundancy factor).

Distribute tasks Each subtask is assigned to a randomly selected participant (worker), but not all subtasks should be processed by one participant, unless he/she is the only one who is active. A subtask is sent from the project owner to the worker, who executes it on his/her local machine. Participant should only accept new subtasks from projects they are subscribed in, other jobs should be ignored. The result of a subtask is sent back to the owner. If the worker could not finish the work in a defined time period (deadline), the project owner reassigns the subtask to another worker and ignores received results which are over the deadline.

Time control Participants are able to block or allow incoming subtasks for registered projects by defining a relative point in time, e.g. now, in five minutes or in two hours.

Incentives Users receive incentives (points) for submitting correct results. The project owner delivers completed subtask to a server, which compares already received subtasks with the same input data and accounts credits for same outcomes. The result which was the most often submitted will be seen as the correct one. The amount of points which a user receives depends on the type of task and the size of data that the user has processed. It should not be possible to manipulate points of any user.

4.2 Non-Functional Requirements

Non-functional requirements describe characteristics that the system should fulfil.

Integrity and authenticity It should not be possible to manipulate data sent between users or to impersonate other users.

Confidentiality A message sent from one user to another must not be readable by an attacker (man in the middle).

Scalability It should be possible that the number of users grows arbitrarily without having to modify the architecture or source code.

Dynamics Users should be able to join and leave the system at any point of time without the need to restart the application.

Usability The GUI must be easy and intuitive to use without the need of reading a manual.

Performance The application should respond in a practical time. Operations that are time-consuming must not block the GUI and have to be executed in a background thread.

Cross-platform software The program should be able to run on multiple platforms to ensure a wide user-base.

4.3 Attack Scenarios

This section considers potential security threats to which the developed application could be vulnerable. Each scenario consists of a brief description, the resulting consequences if the attack was successful and a possible countermeasure that would prevent exploiting the vulnerability.

4.3.1 Man in the Middle

An attacker intercepts messages between two peers.

Consequences The attacker can alter messages, so that the receiving peer would get a message with different data or redirect messages to another peer. For example modification of task results or stealing points.

Possible Solution All messages between peers should be encrypted.

4.3.2 Impersonate other user/company

The attacker gets access to the private key of a user.

Consequences By using the certificate and private key of another user, the attacker is able to receive messages that were intended for the victim.

Possible Solution Private Key must be kept at a secure place, e.g. HSM or smart card.

4.3.3 Deliver wrong results

Participants of a project deliver random results without doing the actual calculation of the subtask.

Consequences The cheater could earn more points in the same time as a regular user, because random results can be returned immediately.

Possible Solution Sending the same subtasks to multiple users and check if the results are equal.

4.3.4 Malicious code

Project owners create a project or task with malicious code (e.g. virus), which will be executed on the participants' machines.

Consequences The attacker could infect many machines in short time and use them for example as a botnet.

Possible Solution Participants should execute the code in an isolated environment (sandbox). Furthermore, code signing can assure that only trusted code is executed.

4.3.5 Altering points

Participants manually edit their points.

Consequences Points would not correlate any more with invested time and machine work done by participants. High scores and ranking lists could be manipulated.

Possible Solution The own points should not be stored on the user's machine, but on the owners of projects he/she is signed on. This means that the project owners are in charge over their participants' points. Alternatively, a server could take responsibility for managing the points of the users.

CHAPTER 5

Design

The PRC application described in this thesis uses the Secure Peer Model as communication and coordination middleware. This means that outgoing requests to and incoming responses from other users are executed over the Secure Peer Model. By using asynchronous communication the application is able to continue doing work and will receive the response as soon as it is available. As this project attaches high importance to security, the communication is encrypted by TLS. Therefore, each user must have a public and private key pair, which are used for encrypting the communication. The public key is saved in a digital certificate, which serves as identification proof. Furthermore, attribute certificates store user attributes (e.g. full name or operating system) and have a shorter validity period than the public key certificate. This allows users to change their attribute values more frequently without the need of renewing their identity certificate. Both certificates must be signed by a CA, which verifies that the provided identity is correct and valid. This check is not part of the thesis and therefore not implemented into the program.

This chapter is structured as follows: First an architecture of the designed system should give an impression how the components of the application work together. The subsequent section describes functional features, like user registration or point submission, in detail. The certificate-based authentication method is explained, in addition to investigating PKC and AC. Finally, an overview of the used Peer Model including the structural design of peers, wirings and rules, is given.

5.1 Architecture

The entire architecture for the application consists of a server and multiple users, who can be workers as well as project owners. Once a user is registered into the system, i.e. has a valid username, there is no need to contact the server any more, except for submitting points and modifying user attributes.

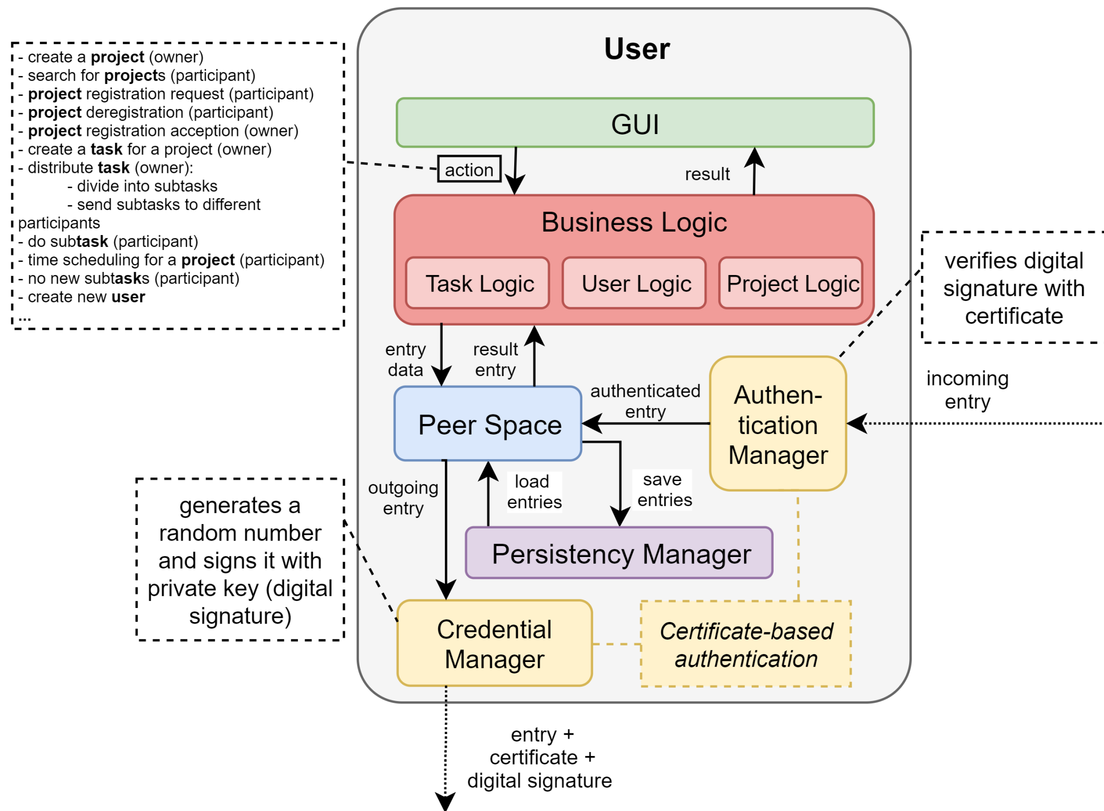


Figure 5.1: Internal architecture of a user.

Figure 5.1 shows the internal structure of a user. At the top the GUI acts as the view layer of the application. It takes actions from the user and delegates them to the business logic, which is separated into three categories: task-, user- and project-logic. The task-logic contains service methods related to task operations like creating a new task or distributing a task to participants. The user-logic is responsible for the user management and allows the registration and signing in of users. Project-related operations, e.g. project search or registration, are encapsulated into the project-logic. The business logic is responsible for initialising peers and wirings via meta-entries. Furthermore, it receives results from the service layer and displays them to the user. Depending on the type of result the GUI decides which layout or data part needs to be updated. A specific action performed by a user in the GUI will result in a corresponding service call in the business logic. All service methods are built on top of the Peer Space, this means that the service wraps an action into an entry and hands it over to the Peer Space. By callback methods the business logic receives an entry as a result, which is converted back into the original data structure. The Peer Space is responsible for delivering an entry to its specified destination (DEST attribute) and for receiving entries. If an entry has an external target address or was sent from another user, then it is going through an authentication process, whereas

internal entries are processed and sent immediately. The authentication manager verifies that an entry belongs to a trusted user and that the stated authentication information is valid. On the other hand the credential manager takes care that outgoing entries are signed with a signature, which is later used by the remote authentication manager for verifying the entry. Both components use public key and attribute certificates to ensure the ownership of an entry and that the user has the right permissions. The detailed process of how this works is described Section 5.2.

The persistency manager is used to save the state of the Peer Model before the user leaves the application. This is achieved by persisting all entries and meta-entries of all peers on disk. When the user logs back into the application, the state is restored by inserting all entries and meta-entries back into the corresponding peers. The persistency manager makes use of symmetric encryption to protect the user data from unwanted access. The data is secured with the login password of the user and is written into a separate file for each user account. The login mechanism uses the persistency files to verify the correctness of the provided credentials. The username is used to identify the file that belongs to the user, whereas the password is used to decrypt the content of the found file. If the decrypted data is valid, i.e. can be parsed into entries of the Peer Model, the login is successful and the persistency manager restores the user data. Otherwise, if no file matches the username or the decryption fails because of a wrong password, the login fails. Certificates are persisted by using user-separated keystores. To ensure the confidentiality of private keys the keystores are also encrypted with the login password of the user.

For a better understanding an example should highlight the underlying mechanisms of the architecture. Let's assume that the user wants to search for a specific project. The GUI creates a project search request action and calls the corresponding service in the project logic. There the action is wrapped into an entry readable for the Peer Space. For each active peer the entry is duplicated in the business logic, whereas the destination property is set accordingly, so that every logged in user will receive the search request. The Peer Space uses the credential manager to sign the entry with a digital signature and integrate the public key and attribute certificates into the entry, which is sent to its destination. Each active user will receive a project search request from the originator, which will be internally processed. The final outcome, a collection of projects owned by the user, is sent back to the originator of the search request. The authentication manager receives the project search responses and makes sure that for each entry the sender is a user owning a certificate signed by the CA server. The Peer Space passes the entry on to the project service layer, where the responses are processed and a result is created. Finally, the GUI displays the outcome readable for the user, in this case it shows a list containing all projects that were found.

5.2 Certificate-based Authentication Component

The Secure Peer Space is capable of authenticating incoming entries. Unfortunately the only currently implemented authentication method, a simple password file protection, is not applicable for real world scenarios, as users must exchange their passwords before they are able to communicate with each other. Moreover, the passwords are only saved in plain-text. Therefore, it is part of this thesis to extend the authentication component with a certificate-based authentication mechanism. It is necessary to implement a *Credential Provider* (component of the *Credential Manager*) and a *Credential Verifier* (component of the *Authentication Manager*) for defining a new authentication method. The first realisation (*Certificate Credential Provider*) is responsible for creating a certificate signature every time an entry is being sent to a remote machine. The *Certificate Credential Verifier* checks that incoming entries have a valid certificate, which was signed by the CA. The communication between the peers is over TLS to prevent man-in-the-middle attacks. It is ensured that users trust only certificates which were signed by the CA and that clients have to authenticate themselves by proving that they are in possession of the private key corresponding to the PKC.

The user data is split into two certificates, an identification (or authentication) certificate and an attribute (or authorisation) certificate. This has the benefit that the authentication data is separated from the authorisation data. Furthermore, the authority can assign different validity periods to both certificates, which is useful as authorisation attributes are going to change quicker than identification properties.

The authentication certificate contains a public key and is therefore also called public key certificate (PKC). Additionally, an identification attribute of the user, the username, is embedded into the PKC. This certificate is used for authenticating a user. For simplicity, the PKCs are also used by the TLS protocol to establish secure communication channels between users.

The attribute certificate (AC) contains all user properties (except of username), which are used for granting access to projects of other users. When a user tries to subscribe to a project, the owner verifies that the attributes from the applicant's AC are in agreement with the project-specific policies. The AC will be automatically reissued once the certificate is expired.

An AC contains the following attributes:

1. Full **name**
2. Is the user a real person or a **company**?
3. **E-mail** address
4. **Date of birth** or foundation date
5. **Country**
6. City
7. Street
8. CPU name

9. Operating system
10. Earned points

The first five properties (indicated by bold text) are mandatory and required for a user registration. The last attribute, earned points, can not be modified by the user. By sharing resources to projects and successfully completing jobs of other users, the CA automatically enters and updates the gained points into the user's AC, in the next reissuing cycle. The detailed verification process of the user attributes done by the CA is not specified in this thesis. It is assumed that the data of the issued certificates is correct.

5.3 Functional Features

In the following essential functionality of the application is presented and explained in detail.

5.3.1 User Registration

The first thing new users must do is to fill out some information about themselves (e.g. name, birth date, address) and specify a username and a password, which is used for the user login and encrypting the persistency data (including the keystore). This data (without the password) is packed together with an automatically generated public key into a CSR and sent to the CA. Before that the Peer Model is initialized using the username as name for the Runtime Peer. All communication happens over TLS, therefore a self-signed certificate is used for the first time a peer contacts the CA server. In case of a user registration request the server will trust all certificates. The provided user data is going through a verification process, which is not part of this work but is required in real world.

For the sake of convenience the server is also used for managing signed up users. Users are uniquely identified by their provided usernames, which are directly linked to their public keys. The CA server refuses registrations with usernames that already exist in the system. If the registration is successful, the data from the CSR is split into two certificates. The username and public key is put into a PKC, whereas the remaining properties are transferred into an AC. This has the advantage of assigning different lifetimes to the certificates. As properties used for authorisation can usually change quickly, ACs have a shorter validity period than PKCs. Both certificates are signed by the CA using the secret private key of the server. At the end the user will either receive a successful response containing signed PKC and AC, or a failed answer if the username is already registered by another user.

The user registration process is depicted in Figure 5.2.

After a successful registration or login the user is able to communicate with other users and has the possibility to create a new project, search for a project or register to a project.

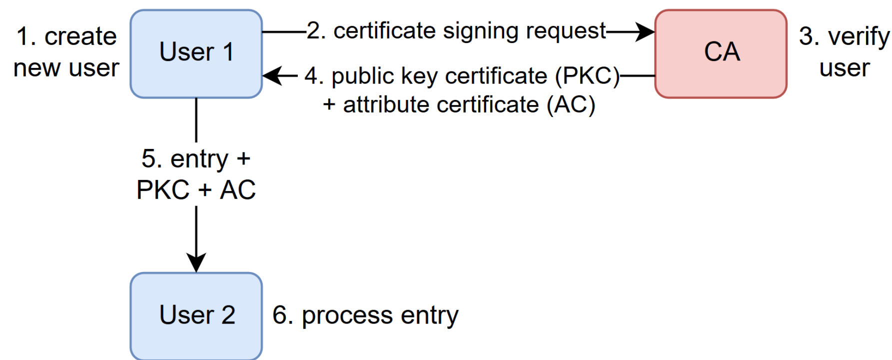


Figure 5.2: User registration process with CA-server.

Furthermore, users can change their attributes by sending corresponding requests to the server. As ACs have a lifetime of some days, these requests are sent automatically by the application shortly before the certificates will expire. The CA verifies the attributes (not part of the thesis) and reissues a new AC. Users are able to change their attributes at any time, but the inscription into the AC is done in the next reissuing cycle.

5.3.2 Project Registration

Users who want to share their resources need first to find a project they are interested in, before they can participate in it. This is done by a search, where all active users are asked to list their projects, from which the requester can choose. The lookup is done through the Peer Model, which keeps track of all online peers (see Section 6.3.2 and 6.3.1). Project owners are able to exclude other users from their projects by defining generic rules (e.g. only users coming from Austria are allowed). For building these rules, all attributes from the user registration can be used. When a user tries to register to a project, the owner takes the user name from the PKC in addition to the attributes from the AC and compares them with the project rules, which are realised with the Secure Peer Model authorisation rules (see Section 5.4.3). If the rule allows the user to participate, a successful response is returned. Otherwise, the registration request is answered with a *not allowed* message. As the rule is already applied to project search requests, users will only see projects they are allowed to register. Due to the possibility that project owners modify the rule after a project search was performed, it is required to reverify the permission of a user who has sent a registration request.

Figure 5.3 shows the sequence of a project registration. $User1$ wants to participate in a project. For this a search request is sent to all other users. They answer with a list of projects $User1$ is able to join. $User1$ decides to contribute to the project of $User5$ and sends a registration request to that user. $User5$ answers with a registration response containing $SUCCESS$ or $NOT_ALLOWED$.

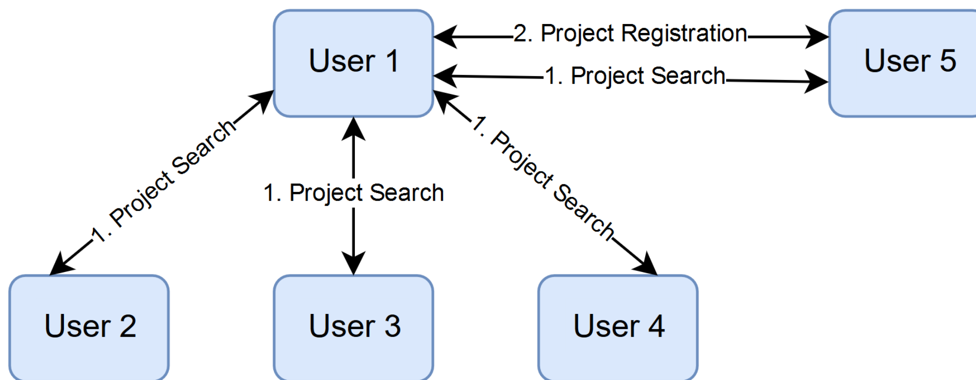


Figure 5.3: Project search and registration process.

5.3.3 Task Distribution

A user will receive new subtasks from registered projects. Unfinished subtasks of a previous session are automatically resumed. Participants transmit the result of successfully completed subtasks back to the project owner. The outcome is saved locally and transmitted to the server, where the worker of the subtask will receive points if the result is correct (see Section 5.3.4). Only one subtask is executed at a time on the participant's machine. It can be stopped and resumed at any time. Furthermore, users are able to block specific projects to which they are subscribed. This means that they will not receive any new subtasks until the project is set to active again. This can be done manually or automatically by setting a relative time on how long the project should be blocked. The same holds for activating a permanently blocked project.

Project owners can decided if a subtask should be executed on multiple machines by choosing a redundancy factor higher than zero. This factor determines how often a subtask should be executed redundantly on different machines. This is useful to compare the outcome of several users and hinder them to submit faked results. For example, if one user tries to earn more points by always returning the same result for all incoming subtasks, the project owner can assign a redundancy factor of two to the task. Then each subtask would be executed by three non-identical users. Assuming that the cheater will receive work from this task and that the other participants always return the correct result, the project owner will receive three results for each subtask, where at least two are the same and one could be from the cheater. The owner can identify the faked results by accepting the most common result as correct and abandon the cheating user from the project. This workflow is not automated and has to be done manually by the project owner.

The task distribution of a redundant subtask can be seen in Figure 5.4, which also includes the subsequent submission of points.

5.3.4 Point Submission

Users are rewarded with points for participating in a project and doing subtasks. For each submitted subtask they will earn a project-specific amount of points. Figure 5.4 describes how users can earn and query points. $ProjectOwner$ is the user who has created a project, which $Participant2$ and $Participant3$ have registered to. First the owner transmits redundant subtasks ($subtask1$) to the participants. They solve the subtasks by spending their computational resources. The results go back to the project owner, who stores them for later usage before submitting the completed subtasks to the CA server. If the subtasks share the same result, the CA gives both participants points. An alternative solution would be that project owners directly validate the results of their participants and only send the points to the CA. This would decrease the server load and would enable an automatic identification of fake results (see also Chapter 8).

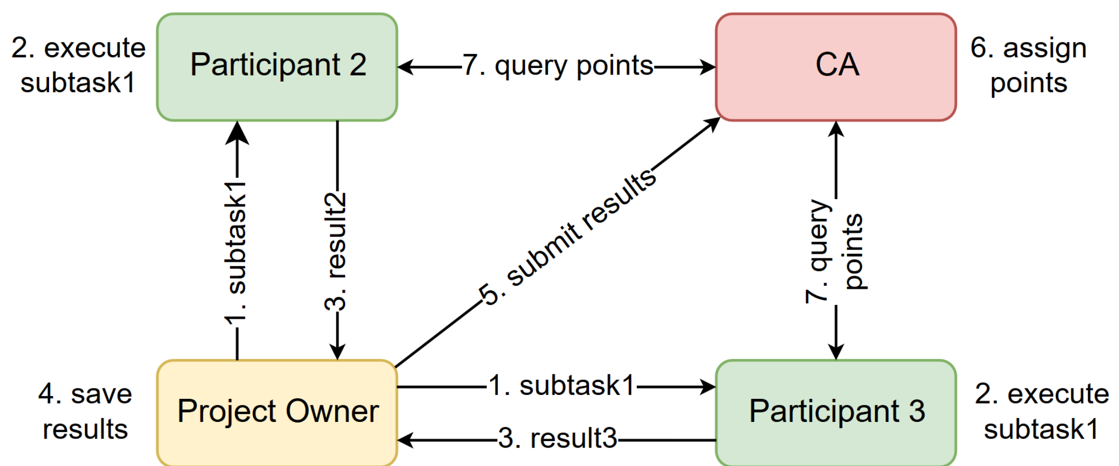


Figure 5.4: Task distribution and point submission processes.

To avoid that participants get points for submitting fake results, the point submission only works with redundant subtasks. When the server receives a completed subtask and there does not exist any subtask that corresponds to the same project and has the same input data yet, a field in the subtask, the point status, is set to $VALIDATING$. As soon as a redundant subtask that has the same outcome arrives from another participant at the CA, the users who solved the tasks will receive points and the status of both subtasks is set to $CORRECT$. If the results do not match, the most common result will be seen as the right solution and the other subtasks will be assigned with the status $WRONG$. In the situation that there does not exist a most common result, all subtasks are set to $UNDETERMINED$. Furthermore, users will only receive points for new subtasks. Submitted subtasks of a user that contain the same input data as an already registered job are marked with a $DUPLICATE$ point status and counted as zero points. This is necessary, as otherwise users only need to calculate the result of a problem once and can then submit the solution multiple times for earning easy points. Moreover, it

does not help project owners to assign the same task to the same user again, when they have already received the result from the user.

Users can see their earned points by sending a point query request to the CA server, which responds with point entries containing beside the actual earned points for each job also information about the completed subtasks and the time of deliverance. The points users receive should correspond to the resources they have spent for solving a subtask. One possibility would be to measure the time they need to solve a task, and use this as basis for the point calculation. The problem here is that users work on different machines and the solving time of a job does not correlate to the used resources. Consequently, another approach is necessary, where the number of points depend on the subtask's input size. Projects differ in the resources needed to solve equally sized tasks. For example, sorting a list with 10 elements is much faster than brute forcing a password on a file with 10 attempts. Hence, each project type has a constant factor that determines how much points a user will earn for executing one data element of a subtask belonging to such a project. The final points the user will receive is the multiplication of this factor with the input size of the subtask that was solved and submitted.

5.4 Peer Model

The designed Peer Model uses several peers, each operating in a different area of responsibility. The $UserPeer$ is capable of creating new users by sending a CSR to the CA. Moreover, it is responsible for the logins of existing user. On the other side, the $UserCAPeer$, running on the CA server, decides if user registrations are valid and sends appropriate responses to the requesters. Furthermore, it manages the information of all users that exist in the system. The $ProjectPeer$ is used to store self-created projects in addition to communicating with other $ProjectPeers$, for example to search for projects of other users. Tasks are managed by the $TaskPeer$, which is able to start and stop tasks together with distributing subtasks to participants. $WorkPeers$ execute subtasks and send the results back to the $TaskPeer$ of the project owner. The completed subtasks will be passed on to the $PointsPeer$, which is located at the CA server. This peer converts subtasks to point entries and stores them for later queries by the user.

In the following, the individual peers as well as their wirings and rules are illustrated and explained in detail. Furthermore, the connections between the peers are demonstrated.

5.4.1 UserPeer

The UserPeer is used to register a new user, to renew expired certificates and to take care of point entries. These processes are depicted in Figure 5.5 in the left upper part as well as their interactions with the CA server ($UserCAPeer$ and $PointsPeer$).

The wiring W_1 takes a user registration response ($URes$) (sent by the CA) from the PIC. It processes the response by calling the $CSRResponse$ service, which persists the new certificates in a keystore. If the registration was successful, the wiring writes a user entry

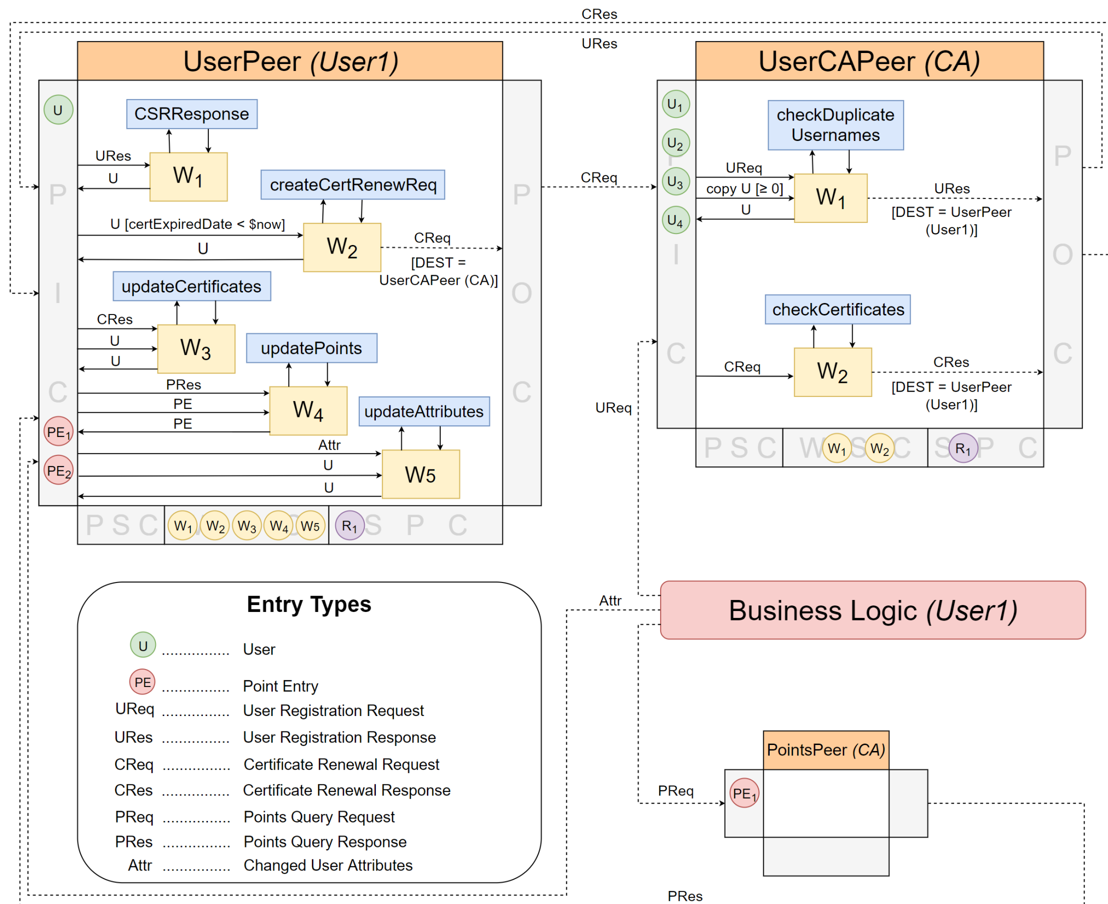


Figure 5.5: Structure of UserPeer and UserCAPeer including wirings and rules.

(U), containing information of the current user, back into the PIC. This entry serves amongst other things for later user identification of a persisted peer model, i.e. after a login. As soon as a certificate (PKC or AC) is expired, the wiring W_2 takes the user entry from the PIC and calls a service to create a certificate renewal request ($CReq$) addressed to the $UserCAPeer$ of the CA server. The $certificateExpired$ date of the user entry is increased by a retry-value and U is written back into the PIC. If the CA should not respond within this timespan, W_2 fires again, whereby the $CReq$ is resent to the CA. The response ($CRes$) is handled by wiring W_3, which substitutes the expired certificates by the newly received ones and updates U with the new $certificateExpired$ date. Point query responses ($PRes$) are consumed by wiring W_4. It writes back newly received point entries (PE) and updates existing ones. These entries represent rewards that the user has received for successfully finished subtasks. The CSR ($UReq$) and the point query request ($PReq$) are manually triggered by user interactions. When the user changes attributes through the GUI, an $Attr$ entry, containing the new attributes, is sent to the PIC of the peer. W_5 takes this entry and updates the corresponding fields in the

U entry.

Security Rules

The rule R_1 allows the CA server to write user registration ($URes$), certificate renewal ($CRes$) and point query responses ($PRes$) to the PIC of the $UserPeer$.

NAME: R_1

SUBJECTS: $[role = CA]$

RESOURCES: PIC

OPERATIONS: $write$

SCOPE: $URes$, $CRes$, $PRes$

5.4.2 UserCAPeer

CSRs ($UReq$) and certificate renewal requests ($CReq$) are managed at the $UserCAPeer$. At the right upper part of Figure 5.5 the peer and its internal structure are shown.

Incoming user registration requests ($UReq$) are consumed by wiring W_1, which makes use of the service method $checkDuplicateUsernames$ to verify that the username from the CSR does not exist in the application yet. If this test succeeds, a response ($URes$) is created containing signed PKC and AC. Furthermore, a new user entry that consists of user data (username, email, ...) is written into the PIC. The wiring W_2 is responsible for verifying the validity period of certificates. It takes a certificate renewal request that contains expired PKC and/or AC. The service function $checkCertificates$ takes care that the delivered certificates are owned by the requester, signed by the CA and are really expired. If all of these conditions are fulfilled, renewed certificates containing the same data are issued and transmitted back to the user.

Security Rules

The rule R_1 allows users to write user registration ($UReq$) and certificate renewal requests ($CReq$) to the PIC of the $UserCAPeer$.

NAME: R_1

SUBJECTS: $[role = User]$

RESOURCES: PIC

OPERATIONS: $write$

SCOPE: $UReq$, $CReq$

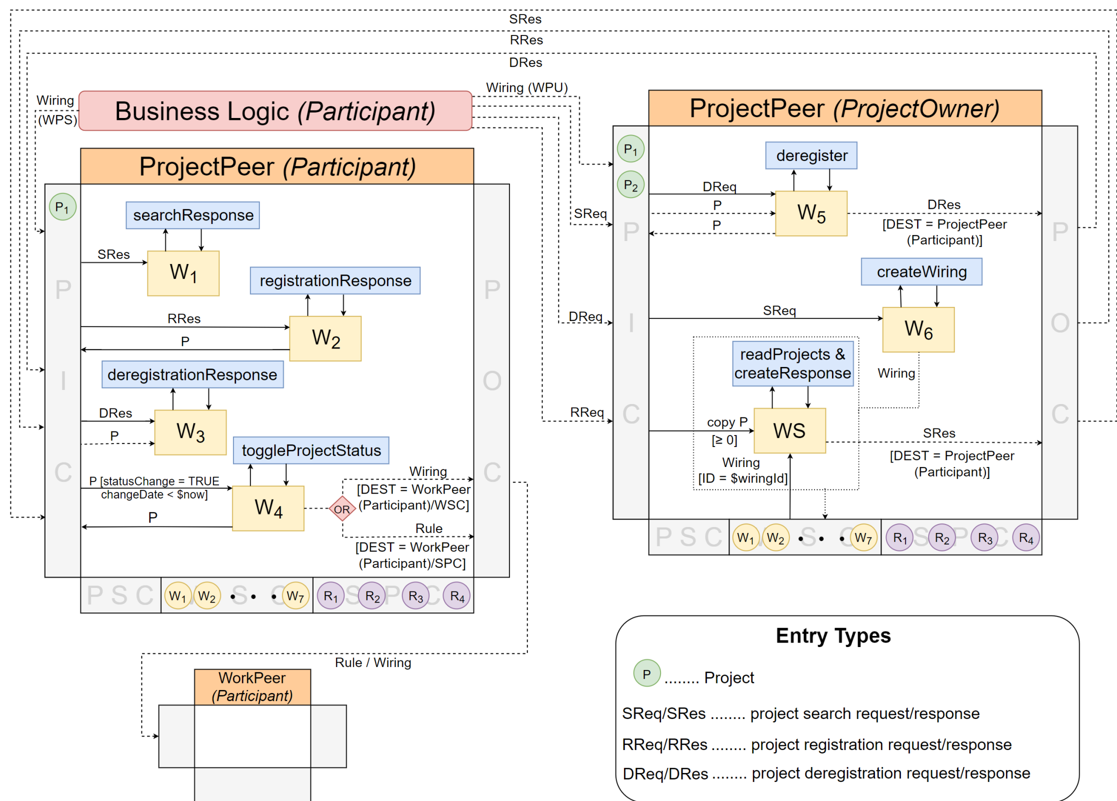


Figure 5.6: Structure of ProjectPeer including wirings and rules.

5.4.3 ProjectPeer

Project entries (P) are organised at the $ProjectPeer$. It is responsible for project searches ($SReq$), project registrations ($RReq$) and project deregistrations ($DReq$). Moreover, the peer allows changing the status of subscribed projects. This means the participant is able to set selected projects online to receive subtasks, or offline to hinder the project owner from delivering new jobs to the $WorkPeer$. These workflows are illustrated in Figure 5.6.

The participant on the left side sends user-initiated requests ($SReq$, $RReq$ and $DReq$) through the $Business\ Logic$ to the $ProjectPeer$ owned by the project owner on the right side. Project search requests ($SReq$) are handled by wiring W_6. It creates a new wiring WS in the name of the requester. The project owner can decide which users are able to see their projects by defining generic rules individually for each project (see security rule R_2). Hence, wiring WS will only return projects (P) that the originator of the request is allowed to read. Before sending the list of found projects back to the user, the wiring removes itself from the WSC as it is only needed once and will be generated again for the next request. When the participant receives a search response entry $SRes$, wiring W_1 will be triggered. The result of found projects will be displayed to the user as

a consequence of executing the service *searchResponse*. Project registration requests ($RReq$) are consumed by wiring W_7, which is not depicted in the graphic as it works similar to W_6. It also creates a dynamic wiring on behalf of the initiator, but instead of reading all possible projects, it only modifies the project which the user wants to register to. If the project exists and the user is allowed to participate, a successful registration response is returned. Wiring W_2 reacts to a successful answer of $RRes$ by saving the project as an entry (P) in the peer. Consequently, users can have two different types of project entries in their $ProjectPeer$, one they participate in and one they own. The owner attribute of a project helps to distinguish between them. If participants want to unsubscribe from a project, they send a project deregistration request ($DReq$) to the owner. Wiring W_5 accepts such requests and calls a service function to deregister the user from the project. A "one-shot" wiring updates the corresponding project entry using direct access. It takes the project, identified by the *projectId* of the $DReq$, from the PIC, removes the user from the participants list, writes the project back and deletes itself from the WSC. The participant's wiring W_3 waits for the response and removes the project entry with the help of a dynamic wiring, which deletes the project from the PIC and itself from the WSC. The dynamic wirings of W_3 and W_5 are simplified by dashed arrows due to little drawing space.

Although project owners can send work to their subscribers, the rule which allows this is still in the hands of the participants. By defining a relative time duration they can toggle the status of a project from $ALLOW$ to $BLOCKED$ or vice versa. An absolute timestamp is calculated by adding the relative to the current time, i.e. a zero relative time means that the change will take place immediately. A project status change is activated by the $Business\ Logic$ via a "one-shot" wiring (WPS). It updates the project entry by switching the *statusChange* property to $TRUE$ and assigning the computed absolute timestamp to the *changeDate* field. When the status change date is reached, wiring W_4 takes out the corresponding project and sets the *statusChange* field back to $FALSE$. The called service toggles the status by adding the rule to or removing the rule (with the help of a wiring) from the $WorkPeer$ on the same machine. Section 5.4.5 describes how the $WorkPeer$ handles a project status change.

The dynamic wiring WPU is sent from the business logic into the WSC of the project owner. It modifies a specific project entry, e.g. changing the registration rules, and is removed after the operation.

Security Rules

Rule R_1 allows users to write project searches ($SReq$), project registrations ($PReq$) and project deregistrations ($DReq$) as well as the corresponding responses to the PIC of the $ProjectPeer$.

NAME: R_1

SUBJECTS: $[role = User]$

RESOURCES: PIC

OPERATIONS: *write*

SCOPE: $SReq$, $SRes$, $RReq$, $RRes$, $DReq$, $DRes$

Rule R_2 states that the originator of a request must be authorised by the project owner. If the project-specific rule, which can be dynamically modified by the owner, evaluates to true, i.e. all user attributes fulfil the constraints of the project rule, the dynamic wirings of W_6 and W_7 are able to access the project lying in the PIC. $RuntimeUser$ is the user who owns the locally running Peer Space.

NAME: R_2

SUBJECTS: $[ID = RuntimeUser]$ **for** $[role = User]$

RESOURCES: PIC

OPERATIONS: *read*, *take*, *write*

SCOPE: $P\ [projectRule.evaluate(\$originator) = TRUE]$

To allow placing the wiring WS into the WSC on behalf of a user who made a $SReq$ or $PReq$ the following rule R_3 is needed. As the wiring removes itself after activation, a *take* operation is also necessary.

NAME: R_3

SUBJECTS: $[ID = RuntimeUser]$ **for** $[role = User]$

RESOURCES: WSC

OPERATIONS: *write*, *take*

SCOPE: $Wiring$

The result ($SRes$ or $PRes$) of wiring WS is written into the POC of the $ProjectPeer$, where it waits for its further delivery to the remote peer of the requester. The wiring is indirectly owned by the user who initiated a project search or registration request. Therefore, all entries the wiring generates are also owned in such a manner. Rule R_4 permits the local user to write a $SRes$ or $RRes$ to the POC in the name of another user.

NAME: R_4

SUBJECTS: $[ID = RuntimeUser]$ **for** $[role = User]$

RESOURCES: POC

OPERATIONS: *write*

SCOPE: $SRes$, $RRes$

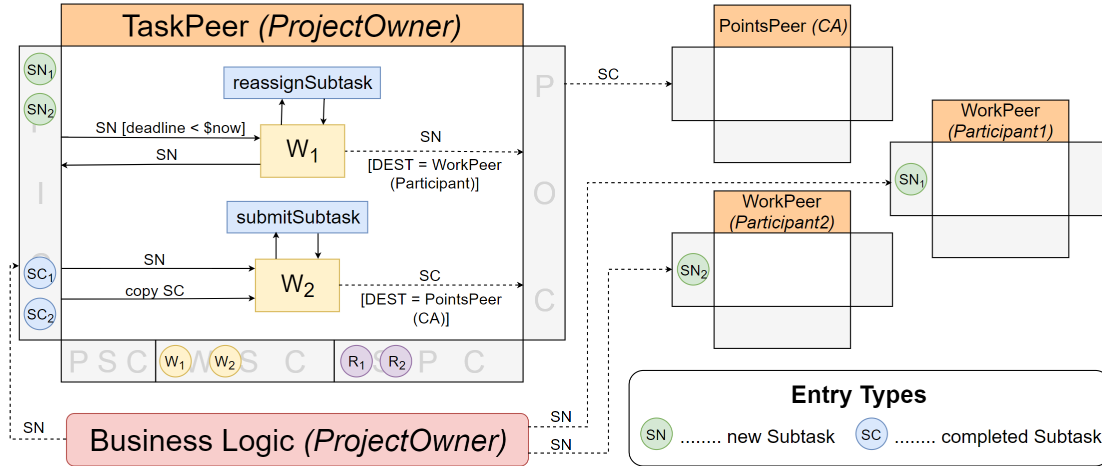


Figure 5.7: Structure of TaskPeer including wirings and rules.

5.4.4 TaskPeer

The *TaskPeer* is responsible for managing tasks and storing new (SN) and completed subtask entries (SC). When a task is started, the work is split into multiple subtasks and distributed to all online participants by using a round-robin procedure. This is done in the *Business Logic* of the project owner. A copy of each transmitted subtask is stored in the PIC of the *TaskPeer*, which is necessary for reassigning an expired subtask to another worker. The *WorkPeer* writes the result included in an SC entry, which is set to the same flow ID as the transmitted SN, back to the *TaskPeer* of the project owner, immediately after the subtask has finished.

As soon as there arrives an SC entry that belongs to the same workflow (determined by flow IDs) as an SN entry, wiring W_2 is fired. It removes the SN entry and hands the completed subtask on to the *PointsPeer* that belongs to the CA. The delegation chain looks as follows: *ProjectOwner* **for** *Participant*. If a subtask is not completed before its deadline, wiring W_1 reassigns and transmits it to another participant. The behaviour of the wirings are depicted in Figure 5.7 with one project owner and two participants.

Security Rules

Writing completed subtasks (SC) to the PIC of the *TaskPeer* is limited to participants who were assigned by the project owner, i.e. the corresponding SN entry must have the originator of incoming SC entry inscribed as assigned worker ($assignedWorker = \$originator.userId$). This is what rule R_1 regulates. The ID of SC is set in the context as variable $\$id$ and must correlate with the ID of SN.

NAME: R_1

SUBJECTS: $[role = User]$

RESOURCES: PIC

OPERATIONS: $write$

SCOPE: SC

CONDITION: $PIC \triangleright SN[id = \$id\ AND\ assignedWorker = \$originator.userId]$

Rule R_2 makes it possible to delegate the SC entry forward to the CA server. For this the local user ($RuntimeUser$) must be able to write the SC in the name of another user (the worker) to the POC.

NAME: R_2

SUBJECTS: $[ID = RuntimeUser]$ **for** $[role = User]$

RESOURCES: POC

OPERATIONS: $write$

SCOPE: SC

5.4.5 WorkPeer

The peer where participants control the execution of subtasks is called $WorkPeer$. Figure 5.8 shows the internal structure of such a peer.

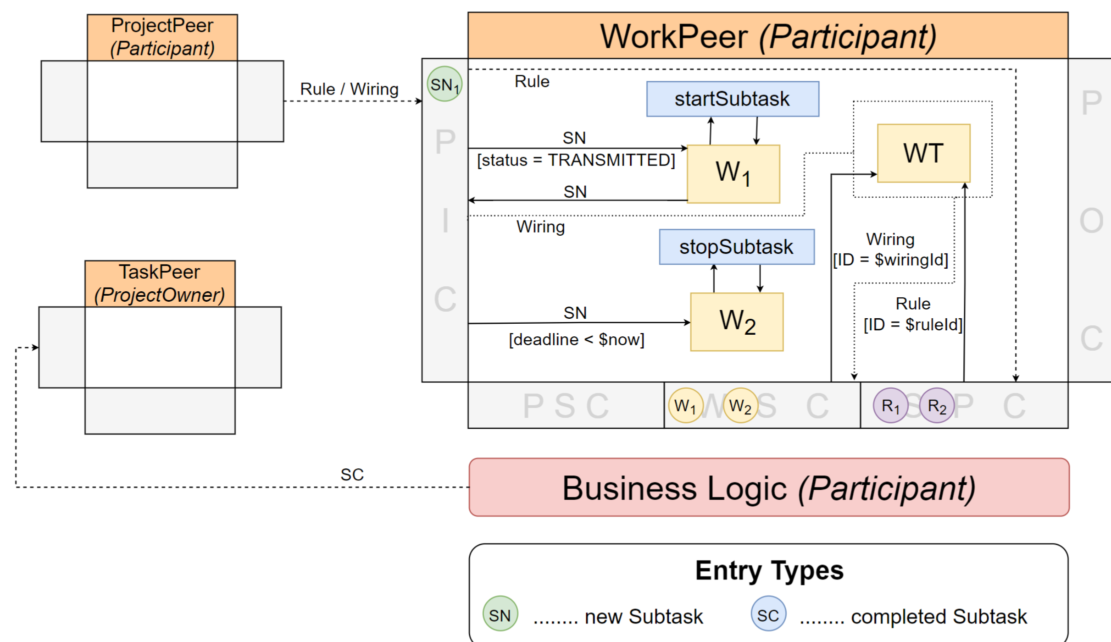


Figure 5.8: Structure of WorkPeer including wirings and rules.

Wiring W_1 listens for incoming subtasks (SN) and executes them. To prevent a concurrent execution of the same subtask the status is changed from $TRANSMITTED$ to $RUNNING$. As only one subtask is allowed to run at the same time, the status is set to $WAITING$ if there is another job running already. When the participant is not able to complete a subtask before it is reaching its deadline (defined by the project owner), wiring W_2 removes the SN entry from the PIC and stops the possible execution of the job. The delivering of SCs to the $TaskPeer$ of the project owner is triggered by the successful termination of the thread which has executed the subtask in the *Business Logic* of the participant. The $ProjectPeer$ sends a rule or a wiring to change the status of a registered project. Rules are added into the SPC of the $WorkPeer$ and allow corresponding project owners to submit jobs to the participant. The dynamically created wiring WT takes such a rule out again to withdraw a previously added permission from the project owner. Moreover, the wiring removes itself from the WSC as it is not needed any more. Rules and Wirings are assigned to uniquely defined identifiers, $\$wiringId$ and $\$ruleId$. This allows for an exact correlation in the guard of the wiring.

Security Rules

For each project, which a user is registered to, a rule RT is necessary. It allows the project owner to transmit new subtasks (SN) to the PIC of the $WorkPeer$. The variable *#project* represents a subscribed project.

NAME: RT

SUBJECTS: $[ID = \#project.owner]$

RESOURCES: PIC

OPERATIONS: *write*

SCOPE: SN $[projectID = \#project.id]$

5.4.6 PointsPeer

The server, where the CA is located, is also used for managing the points earned by users who have successfully completed subtasks. For this purpose an own peer, $PointsPeer$, was created and is illustrated in Figure 5.9.

The project owner submits a completed subtask (SC) that was done by a participant to the $PointsPeer$. Wiring W_2 takes this subtask and converts it into a point entry (PE), which is stored in the PIC. Furthermore, it generates a wiring WPC that is responsible for updating the PEs and calculating the points. For this WPC takes out all PEs of the PIC that are redundant subtasks of the previously arrived SC as well as the freshly converted PE, where the subtask ID matches the ID of the completed subtask. The *redundants* field is already set by the *Business Logic* of the project owner and transferred from the SC into the PE entry by the service of W_2. The variables $\$SC_ID$ and $\$id$ are set by W_2. The first one equals the ID of the incoming SC entry, whereas

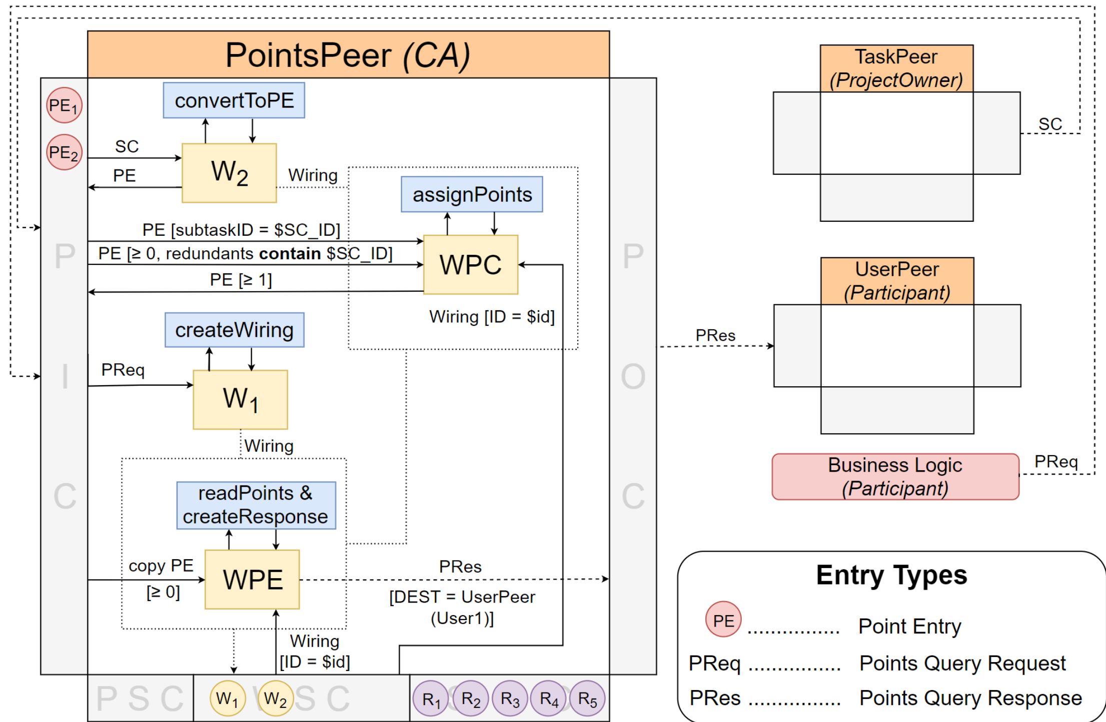


Figure 5.9: Structure of *PointsPeer* including wirings and rules.

the second one is a uniquely created value to identify the wiring WPC. The points are refreshed as described in Section 5.3.4. Afterwards they are written back into the PIC and WPC removes itself from the WSC. Wiring W_1 takes care of point query requests ($PReq$), which are initiated by the *Business Logic* of a participant. With the help of a "one-shot" wiring WPE, the point entries corresponding to the requester are read and put into a point query response ($PRes$).

Security Rules

The rule R_1 allows users to write point query requests ($PReq$) to the PIC of the $PointsPeer$.

NAME: R_1

SUBJECTS: $[role = User]$

RESOURCES: PIC

OPERATIONS: *write*

SCOPE: $PReq$

Using the same principal as R_3 from the $ProjectPeer$, the next rule (R_2) allows to write a wiring of the type WPE into the WSC of the $PointsPeer$ on behalf of another user. The *take* operation is needed, because the wiring is only used for one point query request.

NAME: R_2
SUBJECTS: $[ID = RuntimeUser]$ **for** $[role = User]$
RESOURCES: WSC
OPERATIONS: $write$, $take$
SCOPE: $Wiring$

Rule R_3 permits wiring WPE to write a point query response $PRes$ to the POC.

NAME: R_3
SUBJECTS: $[ID = RuntimeUser]$ **for** $[role = User]$
RESOURCES: POC
OPERATIONS: $write$
SCOPE: $PRes$

Users can only access point entries (PEs) which belong to themselves. This restriction is defined by rule R_4. The *worker* attribute of a PE must match the originator of a point query request.

NAME: R_4
SUBJECTS: $[ID = RuntimeUser]$ **for** $[role = User]$
RESOURCES: PIC
OPERATIONS: $read$
SCOPE: $PE\ [worker = \$originator.userId]$

Rule R_5 allows only users to submit completed subtasks (SC) in the name of another user (*User* **for** *User*). Additionally, the worker attribute of the SC entry must match the originator of the submission.

NAME: R_5
SUBJECTS: $[role = User]$ **for** $[role = User]$
RESOURCES: PIC
OPERATIONS: $write$
SCOPE: $SC[worker = \$originator.userId]$

CHAPTER 6

Implementation

As the design was described in the previous chapter, we now take a look at the implementation of the public resource application. First an overview with a class diagram is given, followed by a presentation of the task model and instructions for writing own task types. The Secure Peer Space was modified to fulfil the requirements of the applications. These extensions are shown and described, after an outline of the former implementation is presented. Afterwards, the interaction with the Secure Peer Space is explained. The final section introduces the GUI and explains its usage.

6.1 Overview

The application was implemented with Java 8, which includes JavaFX. The most important classes as well as their relation to each other are depicted in Figure 6.1 as a class diagram. *PRCMain* is the Java main class and acts as entry point of the application. It decides if the program should be executed with command line interface (CLI) or GUI. The CLI is needed for testing purpose and is not able to perform as many commands as the GUI does. Nevertheless, both are built atop a service interface (*IService*), which holds references to user, project and task service interfaces. These interfaces are realised by implementations that make use of the Secure Peer Space.

6.2 Tasks

The program instructions, which are distributed to participants, are encapsulated into subtask objects. They do not contain executable code, because predefined task types exist in the application. Therefore, it is sufficient to deliver the type-specific parameters to a participant for executing client-side code.

Figure 6.2 shows the task model with the predefined tasks and subtasks. A task belongs exactly to one project and consists of one or more subtasks.

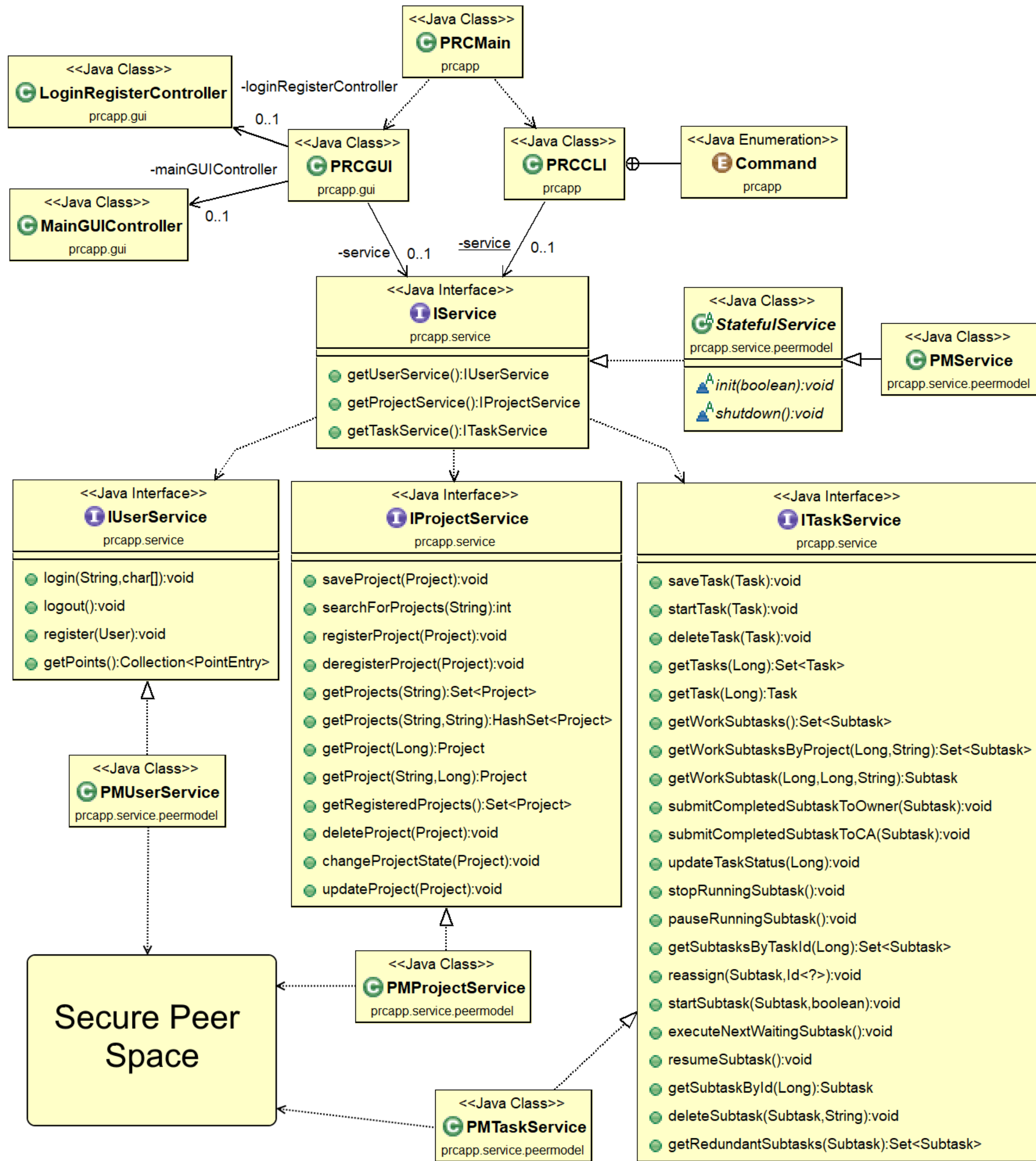


Figure 6.1: Class diagram that gives an overview of the implementation.

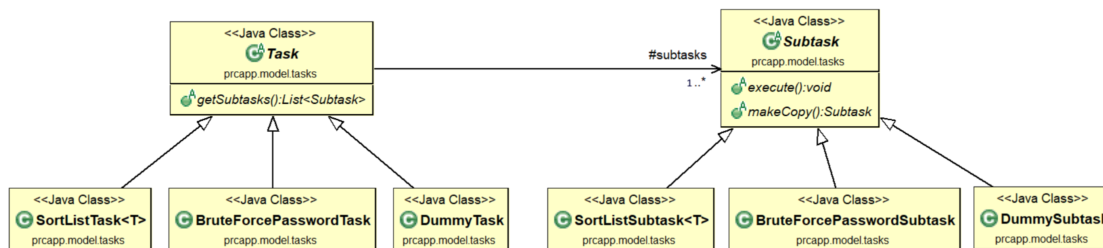


Figure 6.2: Class diagram with tasks and subtasks.

There exist three predefined task types:

SortListTask takes a list of numbers or strings as input data, splits it into equally sized sub-lists and distributes them to the workers. Quicksort is used to sort the sub-lists into ascending order.

BruteForcePasswordTask tries to crack a password of an encrypted file by testing all combinations of possible passwords. The input data consists of an AES-256 encrypted file, a salt and parameters for constructing the password space. As an alternative, an own list with potential passwords can be provided, which would convert the brute force to a dictionary attack. A verifying method checks that the decrypted data is useful and will therefore decide which password is correct. Depending on the use case or scenario the method must be adapted. In this implementation only text files, i.e. containing only printable character (and line breaks), are the defined objective.

DummyTask is the simplest possible task type as it does absolutely nothing. It exists for testing and benchmarking purposes.

All tasks are inherited from an abstract class named *Task*, whereas subtasks share an abstract *Subtask* class. To create a new custom task type, it is only required to implement subclasses for *Task* and *Subtask* as well as overriding the abstract methods.

A short description for these methods is as follows:

Task.getSubtasks(): The first call should generate a list of new subtasks, whereas the following calls only return the generated list. This method normally splits the main problem, defined in the task object, into equally sized sub-problems (subtasks).

Subtask.execute(): By calling this method the subtask will be executed. As it normally contains a time-consuming operation, the implementation calls this method from an own thread. The function should ensure that it is possible to interrupt the ongoing activity, e.g. using *Thread.sleep()* or *Thread.isInterrupted()*. Otherwise, the application is not able to cancel a started subtask. Saving the state of already processed data will allow resuming an interrupted subtask. Calling the

Subtask.setProgress(double progress) method with a number between zero and one allows the user to see how much work has already been done. Caution has to be taken as calling this method too often can freeze the GUI.

Subtask.makeCopy(): This method should make a deep copy of the subtask object. It is used for creating redundant subtasks.

6.3 Adaptations made on the Secure Peer Space

Before describing the changes made on the Peer Space, a short overview of the former implementation is given.

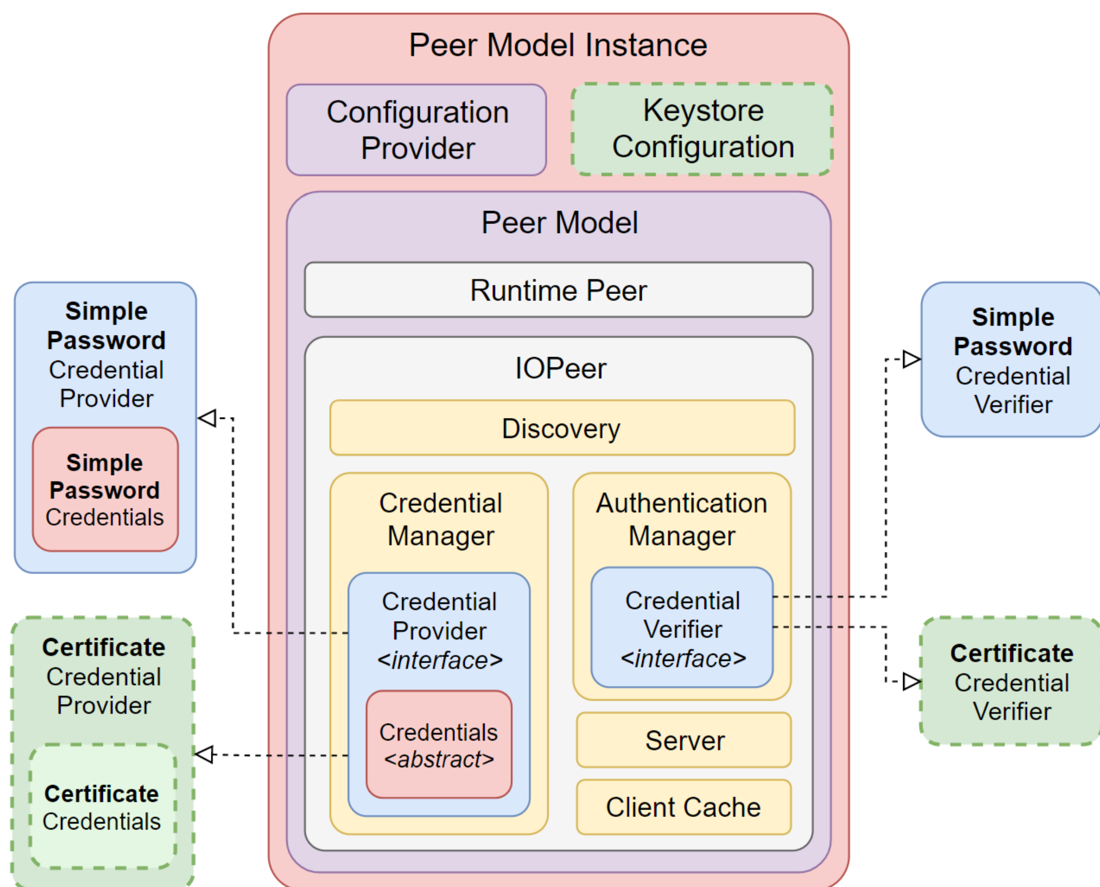


Figure 6.3: An overview of the Secure Peer Space implementation including adaptations.

Figure 6.3 shows the parts of the Secure Peer Space that are relevant for the later explained modifications. Moreover, the graphic includes the new components highlighted by a green background colour and dashed borders. In the following, the individual parts will be discussed shortly.

Peer Model Instance holds a singleton instance of the *Peer Model*. It is also used for initialising the *Configuration Provider* and the *Peer Model*.

Configuration Provider loads user-defined settings from the configuration file into the program.

Peer Model is the core of the framework. The *Runtime Peer* stores entries and meta-entries, i.e also sub-peers. The communication to other peers is handled by the *IOPeer*.

Discovery keeps track of remote *Runtime Peers* and recognises if peers are joining or leaving the system.

Server listens on a specific port for incoming messages.

Client Cache manages *Clients* in a cache for reuse. A client initiates a communication to a remote instance by sending a message to a *Server*.

Credential Manager adds the *Credentials*, retrieved by a *Credential Provider*, to outgoing entries. *Credentials* contain a user ID and a domain.

Authentication Manager uses a *Credential Verifier* to assure that incoming entries from remote peers have correct credentials.

Simple Password Protection was the only implemented security type so far. Each peer has a locally saved file containing attributes (username, password, domain, etc.) of known users. A *Simple Password Credential* of an incoming remote entry additionally contains a password and is verified with the data of the local file, i.e. username, domain and password must match for a successful authentication.

The following components were added to the Peer Space in the course of this thesis:

Keystore Configuration holds information of key pairs used for encryption and authentication.

Certificate-based Protection uses a small PKI implementation in combination with TLS to provide a more secure authentication alternative to the *Simple Password Protection.* A detailed description is located in Section 6.3.4.

It was necessary to extend and integrate the Secure Peer Space by the following adaptations so that the application is capable of providing the required functionality.

6.3.1 Add and remove discovery listeners

Adding custom peer discovery listeners to a running Peer Space enables the possibility to react on peer events (e.g. when a new peer joins the network). It was needed to get notified when a participant comes online and to automatically send unfinished tasks to them. As a *Discovery* instance is already managed in the *IOPeer*, simple delegation methods for adding and removing listeners in the *Peer Model* and *IOPeer* classes were sufficient.

6.3.2 Send to multiple destinations

The Secure Peer Space allows to send an entry to only one peer at a time, but a project search request needs to be sent to all active users. For this reason an additional function was implemented, which transmits an entry to multiple targets. This was achieved by searching for all active peers (using discovery listeners) and sending the entry to each of them separately.

6.3.3 Send with timeout

The discovery of other peers happens in the background of the Peer Space and could take some time, especially after start-up. Due to this fact it was sometimes necessary to wait for other peers to become available, before sending an entry to them. For example the registration of a new user will first start a new Peer Space using the username as peer name and then send a registration request to the CA server. Between these two steps it is required to wait some seconds, until the peer discovery has found the server. The implementation repeatedly tries to send the entry to the given destination until it is successful or the timeout is reached.

6.3.4 Certificate-based authentication

The Peer Space was extended by a further authentication type: certificate-based authentication. In combination with TLS, certificates are used to sign and authenticate entries. A central CA builds the trust basis of the application, i.e. users accept only certificates that are signed by the CA.

The certificate-based authentication is integrated into the Secure Peer Space by implementations of *Credentials*, *Credentials Provider* and *Credentials Verifier*. In addition to the user ID and domain, the *Certificate Credentials* consists of a PKC and a AC. If a UID value exists in the subject's DN of the PKC, this is used as user ID, otherwise a SHA-256 hash value of the public key is generated and set as user ID. The *Certificate Credentials Provider* reads the certificates from the file system and parses them into a *Certificate Credentials* object. Entries from remote peers are authenticated by TLS and by a *Certificate Credentials Verifier*. The first one uses client authentication to ensure that the entity, who wants to send an entry to the user, is in possession of a PKC that is signed by the CA. The *Certificate Credentials Verifier*

checks that the AC is a valid certificate, i.e. is signed by the CA, the reference to the PKC is correct and it is not expired yet. Moreover, the user properties of AC are converted into security attributes, which are later used by the attribute access rules, and stored in a *Principal* instance.

6.3.5 TLS communication

To achieve the integration of TLS into the Secure Peer Space, the communication of the Peer Space had to be adapted so that all incoming and outgoing connections are encrypted. This was done by modifying the *Client* and *Server* classes in the *communication* package. A server socket makes a peer reachable to other peers by opening and listening to a specific port on the computer. Listing 6.1 shows the creation of such a socket in two ways, for encrypted and unencrypted connections. This decision can be set in the configuration file of the Peer Space.

```
private void init() throws IOException, TTransportException {
    KeyStoreConfig keyStoreConfig = PeerModelInstance.getKeyStoreConfig();

    if (keyStoreConfig == null) {
        // use unencrypted communication
        this.serverSocket = new ServerSocket();
        this.serverSocket.bind(null);
        processor = new PMCommunication.Processor<>(handler);
    } else {
        // use TLS protocol
        try {
            serverTransport = getSSLServerSocket(keyStoreConfig);
        } catch (GeneralSecurityException e) {
            e.printStackTrace();
        }
        serverSocket = serverTransport.getServerSocket();
        processor = new SSLProcessor<>(handler);
    }

    new Thread(this::runServer).start();
}
```

Listing 6.1: Server.init() method in peer communication package

If using the TLS protocol, a *KeyStoreConfig* object must be provided. It contains a key store holding public and private keys for the encryption, two trust stores consisting of root certificates that are accepted by the client socket for sending entries and by server sockets for receiving entries and an attribute certificate, which is not relevant for TLS. The *getSSLServerSocket* method generates an Apache Thrift TLS server socket. At the end the server is started in an own thread.

6.3.6 Modifications on configuration file

The Peer Space uses a configuration file for defining peer-specific settings, like space domain or number of threads. As the Secure Peer Space was extended by features that

allow different parameters, the configuration file was enlarged by the following elements:

tlsEnabled By setting this to *true* all communication is routed over TLS.

caServerDomain Multiple CA servers can exist in the system. Therefore, it is necessary to uniquely identify a CA by its domain.

securityProtectionType If security is enabled this attribute allows choosing between *simple password* and *certificate*-based protection.

At the moment it is not possible to send entries from a peer that has enabled TLS to a peer which has not. This limitation exists due to the fact that TLS and Hypertext Transfer Protocol (HTTP) sockets are not compatible to each other.

6.4 Interaction with the Secure Peer Space

All interactions between the GUI and the Peer Space are performed by the business logic. The Peer Model implementation of the service interface, called *PMService*, manages the business logic by controlling the associated sub-services, *PMUserService*, *PMProjectService* and *PMTaskService*. GUI commands are executed by sending appropriated entries to a target peer from a particular service, e.g. logins and registrations are handled by the *PMUserService*. If a user registers to the application, the *PMService* generates the initial peers, wirings and rules (see Section 5.4) by adding corresponding meta-entries to the Runtime Peer. All entries (including meta-entries) are stored in a user-separated persistency file when the user exits the application. This data is reinserted into the Peer Model by the *PMService* after a successful user login.

Wirings and rules are implemented by using the Secure Peer Space middleware. Listing 6.2 shows the realisation of the wiring W_2 from Section 5.4.4. The first guard reads a *Subtask Completed* (SC) entry from the PIC, whereas the second guard takes a *Subtask New* (SN) from the PIC. With the last parameter of a guard the workflow dependency can be set. In the illustrated case both arguments are set to *true*, i.e. the completed subtask and the new subtask must be in the same workflow, which is identified by a flow ID entry field. If the guards are fulfilled, a service method is called by a callback class that implements the *ServiceExe* interface of the Peer Space. This interface consists of an execute function, which is called when the wiring is triggered. The service invokes the callback with the SC entry that is obtained by the service's guard. The callback creates a copy of the completed subtask entry by using indirect access, sets the DEST property of the copy to the *PointsPeer* of the CA server and returns this entry. By the actions of the service and the wiring the entry is transferred to the POC of the *TaskPeer*. There the Secure Peer Space delivers the entry to the defined target peer.

```
pm.addEntry(new WiringEntryBuilder(WIRING_SUBTASK_WORK_DONE)
    .guard(EntryType.getEntryType(PeerConstants.TYPE_SUBTASK_COMPLETED),
        Address.PICAddress, Link.LinkOperation.READ, LinkQuery.ALL,
        LinkCount.EXACTLY_ONE, true)
    .guard(EntryType.getEntryType(PeerConstants.TYPE_SUBTASK_NEW),
        Address.PICAddress, Link.LinkOperation.TAKE, LinkQuery.ALL,
        LinkCount.EXACTLY_ONE, true)
    .service(
            new WiringEntryBuilder.ServiceBuilder(CALLBACK_SUBTASK_WORK_DONE,
                SubtaskCompletedCallback.class)
                .guard(EntryType.getEntryType(PeerConstants.TYPE_SUBTASK_COMPLETED),
                    Link.LinkOperation.TAKE, LinkQuery.ALL, LinkCount.EXACTLY_ONE)
                .action(EntryType.getEntryType(PeerConstants.TYPE_SUBTASK_COMPLETED))
            )
    .action(EntryType.getEntryType(PeerConstants.TYPE_SUBTASK_COMPLETED),
        Address.POCAddress, LinkOperation.TAKE, LinkQuery.ALL, LinkCount.ALL)
    .dest(new Address(new Address.PeerAddress(PeerConstants.TASK_PEER),
        Address.WSCAddress))
    .build());
```

Listing 6.2: Wiring that submits subtasks to CA

The rule R_3, which allows taking and writing wirings from the *ProjectPeer* (see Section 5.4.3), is depicted in Listing 6.3. The variable *runtimeUserForUser* is a subject template that defines the delegation chain: "$[ID = RuntimeUser]$ **for** $[role = User]$". The *EntryTypePredicate* class that is defined in the scope of the rule is used to match the names of entry types, i.e. the rule applies only to (meta-)entries that are of the type wiring.

```
pm.addEntry(new RuleEntryBuilder("AllowTakeWriteFromToProjectWSC")
    .subject(runtimeUserForUser)
    .operation(Permission.WRITE)
    .operation(Permission.TAKE)
    .resource(Address.WSCAddress)
    .scope(new Scope(new EntryTypePredicate(EntryType.WIRING_TYPE.getName())))
    .dest(new Address(new Address.PeerAddress(PeerConstants.PROJECT_PEER),
        Address.SPCAddress))
    .build());
```

Listing 6.3: Rule that allows to write and take wirings from *ProjectPeer*

A special case represents the project rule (R_2 from Section 5.4.3), depicted in Listing 6.4. It defines an entry predicate (shown in Listing 6.5) in the scope field of the rule to decide which projects can be accessed by the subject.

```java
pm.addEntry(new RuleEntryBuilder("AllowReadFromProjectPIC")
    .subject(runtimeUserForUser)
    .operation(Permission.READ)
    .operation(Permission.TAKE)
    .operation(Permission.WRITE)
    .resource(Address.PICAddress)
    .scope(new Scope(originatorBasedProjectAccess))
    .dest(new Address(new Address.PeerAddress(PeerConstants.PROJECT_PEER),
        Address.SPCAddress))
    .build());
```

Listing 6.4: Project rule (R_2) from Section 5.4.3

First the *originatorBasedProjectAccess* entry predicate checks that the investigated entry is a project entry. The access to other types is denied by the predicate (returns $false$). The project, saved as user coordination data in the entry, contains the generic registration rules which are dynamically defined by the project owner. These rules are combined in a *GlobalRule* by using corresponding AND and OR conjunctions (specified by the project owner) to connect the rules together. A boolean value in the global rule defines if users that are matched by the registration rules are allowed or not allowed to participate in the project. Each rule is implemented as an instance of the *AttributeRule* class and contains a *UserAttribute* Enumeration (name, country, points, ...), a *Comparator* Enumeration ($>$, $<$, $=$, $\neq$ or a regular expression) and a $Value$ Object. The creation of the global rule and its user attribute rules takes place in the GUI.

```java
private final static EntryPredicate
    originatorBasedProjectAccess = new EntryPredicate() {
    private static final long serialVersionUID = 1L;
    @Override
    public boolean test(Entry entry, SecurityContext context) {
        if (entry.getEntryType().getName().equals(PeerConstants.TYPE_PROJECT)) {
            Project project =
                (Project) entry.getUserCoData(PeerConstants.PROJECT_DATA);
            if (project != null) {
                Principal originator = context.getSubject().getOriginator();
                return project.getRegistrationRules()
                    .evaluate(MiscUtils.toUser(originator));
            }
        }
        return false;
    }
};
```

Listing 6.5: Entry predicate of the project rule

A recursive evaluation of all rules, starting at the global rule and going down to each attribute rule, is used to determine the outcome of the *originatorBasedProjectAccess* entry predicate by comparing the attributes of the originator from the security context with the defined attribute rules. The *UserAttribute* is used to retrieve the correct attribute value of the originator. The comparison is done by the individual *Comparators*, which take the attribute value and compare it with the $Value$ of the *AttributeRule* by using specific evaluation functions, e.g. the comparator $=$ uses *Objects::equals*. The

GlobalRule makes use of the AND and OR conjunctions to resolve the final result, which is either $true$ or $false$, i.e. the originator is able or unable to access the investigated project. As the entry predicate is defined in the scope of the rule, all entries contained in the PIC are tested against the predicate. Consequently, the access to all projects is controlled by the project rule. Due to the fact that project owners are able to dynamically change the access policies of their projects, it is very laborious to build the project rule by means of simple Secure Peer Space rules. The dynamic part is implemented by the evaluation of the global registration rule in the scope's entry predicate.

The PRC application creates data (users, projects, tasks, . . .) and uses the Peer Model for sharing it with other machines. The information is encapsulated into entries before sending it over the network. For the sake of simplicity the Peer Model is also used for saving these data locally, instead of using a database. Unfortunately the Peer Space did not provide any functionality to persisting peer entries to disk. For this reason a simple persisting extension was implemented, which is capable of saving/loading all entries of a provided peer to/from a file. In addition, the generated files are used for user logins. So if users sign out of the application, all their data are saved to user files, distinguished by the username, and encrypted with AES using the account's password. To sign in to an existing user account, the application tries to decrypt the user file corresponding to the provided username with the entered password. Hence only someone who knows the right password can log into the user account and has access on the data. To prevent multiple logins of the same user, an exclusive lock for the user file is acquired as soon as the user signs in. If another user tries to log in under the same username, the file can not be opened because the lock is already held by the other user.

The persisting extension was wrapped around the Peer Model by using utility functions taking the peer object as parameter and using it to gather all entries for creating a serialize-able data structure that could be saved as a file.

6.5 Application Usage

This section gives instructions for compiling and running the program, followed by a description of the GUI and some screenshots that should help the user to perform the basic operations and become familiar with the application.

The program is built with the aid of Apache Maven [Apa17]. A *readme* file in the root folder contains all commands required for building and executing the software as well as performing JUnit and benchmark tests. Notice that for registering a new user or submitting points, it is required that exactly one CA server is running.

The first screen the application displays after booting is the login and registration window. If users are in possession of an account, they are able to login with their credentials. Otherwise, they must fill in a user registration form, depicted in Figure 6.4, to use the application. The screenshot was made after the registration request was sent to the server, indicated by the animated waiting circle in the middle of the window. The fields which

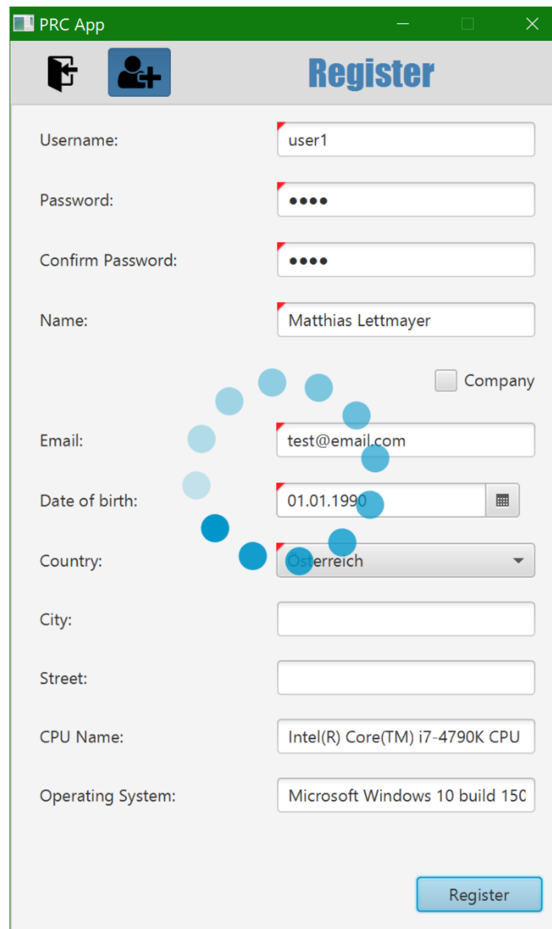


Figure 6.4: Screenshot of the registration form.

are not allowed to be empty are represented by a red triangle in the left upper corner. The *CPU Name* and *Operation System* are read-only fields and are automatically set by the program. Apart from the username, all user attributes can later be modified in the course of a renewal request of the AC.

Supposing that the registration was successful, the user will be automatically logged in and a new window, containing side and top navigation bars, with the *Project Home* view will be opened. This scene is divided into three sections. The first shows all projects the user has created. By right clicking on a project, the user is able to edit or delete it, whereas a left-click reveals all associated tasks in the middle view. The bottom section displays subtasks, corresponding to a selected task, with their status and possible outcome. Figure 6.5 shows a picture of the *Project Home* view, containing two projects. A *SortList* task is selected showing the corresponding subtasks at the bottom. As this task has a redundancy factor of one, each subtask was transmitted two times to different users. The redundant subtasks are grouped together to improve the readability.

PRC App

Projects

Projects

	Name	Type	Rules	Participants
1	BigListSorting	Sort List	Block: All, Allow: (Username = ...	[max, user2]
2	Crack Passwords	Brute Force Password	Allow: All	[user2, max]

Tasks

	Name	Data	No. of Subtasks	Redundant	Timeout	Status
1	EncryptedText	encryptedFile...	10	1	PT1H	RUNNING

Subtasks

Id	Data	Current Worker	Workers	Status	Deadline	Result
[1, 2]	encryptedFile=C...	[user2, max]	[user2, max]	SUBMITTED_CA		
[3, 4]	encryptedFile=C...	max	[user2, max]	INCOMPLETE		
[5, 6]	encryptedFile=C...	[user2, max]	[user2, max]			
5	encryptedFile=C...	max	[max]	TRANSMITTED	03.01.2018 21:05:28	
6	encryptedFile=C...	user2	[user2]	SUBMITTED_CA	03.01.2018 18:56:37	No pass...
[7, 8]	encryptedFile=C...	max	[user2, max]	INCOMPLETE		
[9, 10]	encryptedFile=C...	max	[user2, max]	INCOMPLETE		
9	encryptedFile=C...	max	[max]	TRANSMITTED	03.01.2018 21:05:28	
10	encryptedFile=C...		[user2]	NO_RESULT	03.01.2018 21:05:28	
[11, 12]	encryptedFile=C...	max	[user2, max]	INCOMPLETE		
[13, 14]	encryptedFile=C...	max	[user2, max]	INCOMPLETE		
[15, 16]	encryptedFile=C...	max	[user2, max]	INCOMPLETE		
[17, 18]	encryptedFile=C...	max	[user2, max]	INCOMPLETE		
[19, 20]	encryptedFile=C...	max	[user2, max]	INCOMPLETE		

Figure 6.5: Screenshot of the *Project Home* view.

By clicking on the left-sided arrow in the *id* column, the group can be opened to see the status of the individual redundant subtasks. For example, it can be seen in the screenshot that the subtask with the id 5 was sent to the user *max* and no result has been received yet. On the other hand, the identical subtask 6 was transmitted to user $user2$, who has already solved the job and submitted an outcome to the project owner. The status *SUBMITTED_CA* means that the result of the subtask was transferred to the CA and represents the last step of a subtask's life-cycle. Subtask 10 has the status *NO_RESULT*, which denotes that the job has no outcome yet and currently no participants are working on it (indicated by the empty *Current Worker* column). By clicking on the *retry* symbol on the right side, the subtask will be manually reassigned. Normally this is not required as the work will be reassigned automatically if another contributor is available. The *Workers* column contains $user2$, which states that either subtask 10 could not be transmitted to $user2$ or that the completed subtask was not

returned before the deadline. As the deadline lies in the future, the former holds true.

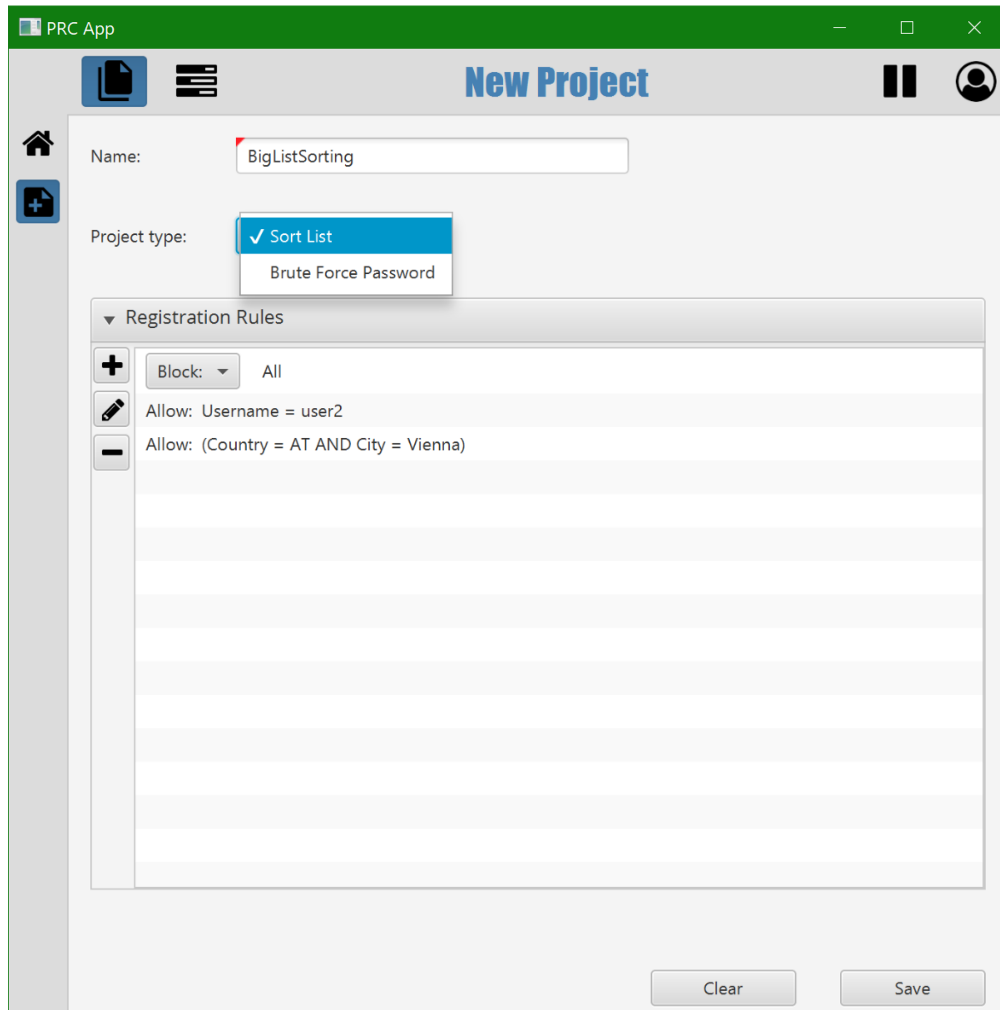


Figure 6.6: Screenshot of creating a new project.

New projects can be created by clicking on the sheet with the plus symbol (bottom icon in the sidebar). This should open the view demonstrated in Figure 6.6. Besides the name and project type, the owner can create rules with the help of generic expressions that decide which users are allowed to contribute to the project. In the graphic all users are blocked, except $user2$ and all users that come from Vienna. Once the save button is pressed, the project becomes visible for other users.

To create a new task, the user has first to open the *Project Home* tab and select a project. If no tasks exist yet, an *Add Task* button is shown in the *Tasks* section. Otherwise, right clicking in the *Tasks* area opens up a context menu, which has a *New* entry that allows to create a new task.

Figure 6.7: Screenshot of generating a new brute force task.

Figure 6.7 shows the window of creating a new task belonging to a *Brute Force Password* project. Besides the general data, an encrypted file as well as a salt must be provided for cracking passwords. Moreover, a list that contains password candidates needs to be generated or specified (dictionary attack).

Participants can use the *Contribution Home* tab to see all their projects they are participating in by clicking on the icon with the bars in the topbar. Each project is shown by its name, type, owner and status. By right-clicking on a project the contributor is capable of unsubscribing from the project or to change the status of the project. Additionally, the view displays information about subtasks that arrived from project owners and allows to control their execution by using the buttons on the right side. Subtasks are automatically executed by the application in FIFO order. The buttons allow the user to manually stop or start another subtask, whereas only one subtask can

Figure 6.8: Screenshot of the *Contribution Home* view.

be executed at a time. Figure 6.8 depicts a screenshot of the *Contribution Home* view, where the user is registered to two projects and has diverse subtasks in the work area. It can be seen in the *Status* column that new work for project *BigListSorting* is blocked until a specific time in the future.

By clicking on the button in the top right corner, symbolised by a user icon, a menu appears showing the full name and username of the currently logged in user. Furthermore, buttons in the bottom allow users to see their points (left), to change their user attributes (middle) and to logout of the application (right). An example of the *User Menu* can be seen in Figure 6.9.

The screenshot illustrated in Figure 6.10 shows the *Point Statistics* page. Each completed subtask that was submitted to the CA is listed here. If the result of a job is correct, appropriate points are rewarded. The status *VALIDATING* denotes that not enough subtasks exist at the CA server to determine if an outcome is valid. The user has to

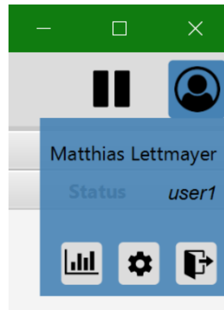


Figure 6.9: Screenshot of the *User Menu*.

wait for other users submitting a redundant subtask that confirms the correctness of the work by an identical result. The *Submitted at* column represents the point in time when a subtask arrives at the server. In the first header of the points view the total points are displayed. The first number (0 in the example) indicates the earned points that are signed by the CA and inscribed into the AC of the user. The second number enclosed by brackets (24233 in the example) represent the up-to-date total points that are not yet written into the certificate and are therefore not visible to other users (e.g. in the course of a project registration).

PRC App

Statistics

Points Total: 0 (24233)

Project / Task	Points	Submitted at	Status
BigListSorting	10		
listA: 23	1	03.01.2018 18:40:17	CORRECT_RESULT
listA: 28	1	03.01.2018 18:03:34	CORRECT_RESULT
listA: 40	1	03.01.2018 18:03:35	CORRECT_RESULT
listA: 38	1	03.01.2018 18:40:17	CORRECT_RESULT
listA: 25	1	03.01.2018 18:03:35	CORRECT_RESULT
listA: 22	1	03.01.2018 18:03:29	CORRECT_RESULT
listA: 34	1	03.01.2018 18:40:17	CORRECT_RESULT
listA: 35	1	03.01.2018 18:40:17	CORRECT_RESULT
listA: 30	1	03.01.2018 18:40:17	CORRECT_RESULT
listA: 32	1	03.01.2018 18:03:32	CORRECT_RESULT
Crack Passwords	24223		
EncryptedText: 6	0	03.01.2018 18:40:17	VALIDATING
EncryptedText: 2	24223	03.01.2018 17:05:08	CORRECT_RESULT

Figure 6.10: Screenshot of the point statistics view.

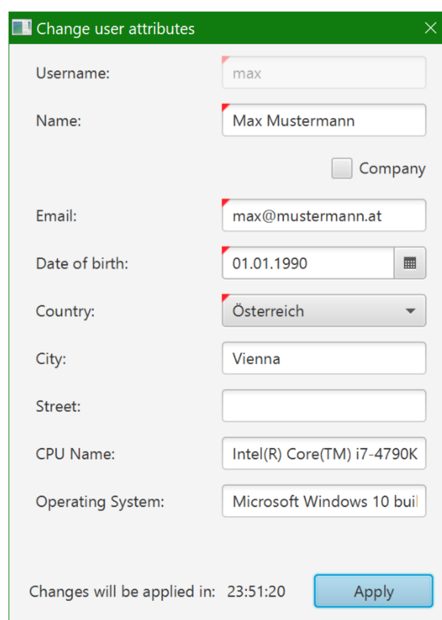


Figure 6.11: Screenshot of changing user attributes.

ACs are issued with a very short lifetime (24 hours). A certificate renewal request is automatically sent to the CA as soon as the AC is expired. Users are able to change their attributes by using the *User Menu*. Figure 6.11 demonstrates a picture of the *User Attributes* form. As the *Username* is used for identifying the user and stored in the PKC, it can not be changed any more. *CPU Name* and *Operating System* are set by system functions and are read-only textfields. The other attributes can be modified by the user. The adjustments will be committed with the next certificate renewal cycle, shown by the timer at the bottom.

CHAPTER 7

Evaluation

The chapter begins with an explanation how the formerly defined requirements are met by addressing the specific parts from the design and implementation. A subsequent use case shows that the application was implemented properly, whereas a benchmark test analyses the performance of two program functions and the underlying Secure Peer Space. Afterwards, a comparison between the implementation and the other systems of the related work chapter is given. This chapter is concluded by a critical reflection about the application and the Peer Model.

7.1 Fulfilled Requirements

In order to show that the application was correctly designed and implemented by the means of previously formulated criteria from Chapter 4, this section explains how these requirements are satisfied.

Create projects Projects can be created through the GUI and are stored as entries in the Peer Model.

Search for projects A search request with a query is sent to all active users with the help of the Peer Model. The users respond with a result matching the query of the search initiator.

Project registration Using the Peer Model, a project registration request is sent to the owner of the project, which answers with a successful registration response if the user is allowed to participate in the project. Otherwise, the project owner replies with a failed registration response. This check is done with security rules of the Secure Peer Model. For a more detailed description see Sections 5.3.2 and 5.4.3.

Create tasks Project owners can create tasks through the GUI by selecting an existing project, entering the name of the task, the number of subtasks, the redundancy factor and other task-specific data. The task is saved as an entry in the Peer Model.

Distribute tasks A task is split into subtask and sent to active participants using the Peer Model.

Time control When the user blocks a project, the corresponding security rule, which allows the project owner to send subtasks to the participant, is deactivated. On the other side if the user activates a blocked project, the security rule is activated again.

Incentives For simplicity the CA server acts also as point server and allows subtask submissions by project owners in the name of their workers. Section 5.3.4 explains this process in more detail.

Integrity and authenticity X.509 certificates ensure authenticity, whereas digital signatures guarantee that messages are not altered. Both concepts are provided by using the TLS protocol.

Confidentiality This requirement is also met by TLS, which encrypts messages between peers and establishes thereby a confidential communication channel.

Scalability By using a peer-to-peer structure the application is able grow arbitrarily. The only bottleneck could be the CA server which handles user registrations and point submissions. In Chapter 8 a possible solution to this problem is given as a future task.

Dynamics The application is able to dynamically integrate and release users by using a peer-to-peer structure.

Usability The GUI was build with JavaFX. The navigation is handled by a top-bar and a side-bar, which change the main view of the application. Self-explaining icons are used to represent the functionality of most buttons.

Performance The performance will be evaluated in Section 7.3 by performing a benchmark test.

Cross-platform software Java 8, which supports Solaris, Linux, OS X and Windows, is used as programming language, whereas JavaFX is not supported on all platforms, e.g. Solaris [Ora17].

7.1.1 Protections against Attack Scenarios

In the following it is explained how the application was protected against the presented attack scenarios from Section 4.3.

Man in the Middle TLS is used to encrypt all communications between the peers (including the server).

Impersonate other user/company The private keys of the users are stored in separate keystores that are protected by the password that is also used for login.

Deliver wrong results Redundant subtasks are used to identify fake results of participants.

Malicious code As the application uses predefined task types, project owners are not able to upload custom code to their participants.

Altering points The CA server manages the point entries of users, which can only see their own points. Participants can only earn points by completing jobs from a project owner, who submits them in the name of the worker to the server.

7.2 Use Case

Previously it has been shown how the requirements of the application are theoretically fulfilled. This section proves the functionality of the implementation by describing a showcase that should also emphasize the practical use of the application.

7.2.1 Setup

An example for using the application is the attempt of deciphering an encrypted text file to know how long it takes to find the right password. Based on the findings the researcher is able to derive the security of the used password and how much effort is needed to crack it.

The setup consists of one researcher, offering a *BruteForcePassword* project, and several users who want to share their resources. The project owner wants that only users with a high reputation are able to participate in the project. This decreases the probability of fake results. Moreover, some exceptions should be possible, so that trustworthy users can be added to the project.

The system setup consists of five users and one CA. In the following the users and their relevant attributes are described.

The scientist who wants to analyse password strengths is named “researcher”, comes from Austria and has zero points.

Username researcher
Country Austria
Points 0

The other users have different points and countries.

Username User1
Country Germany
Points 1050

Username User2
Country Austria
Points 0

Username User3
Country Austria
Points 1200

Username User4
Country USA
Points 1500

It is assumed that $User1$, $User3$ and $User4$ are already registered to the application and have earned their points by contributions to other projects.

7.2.2 Process

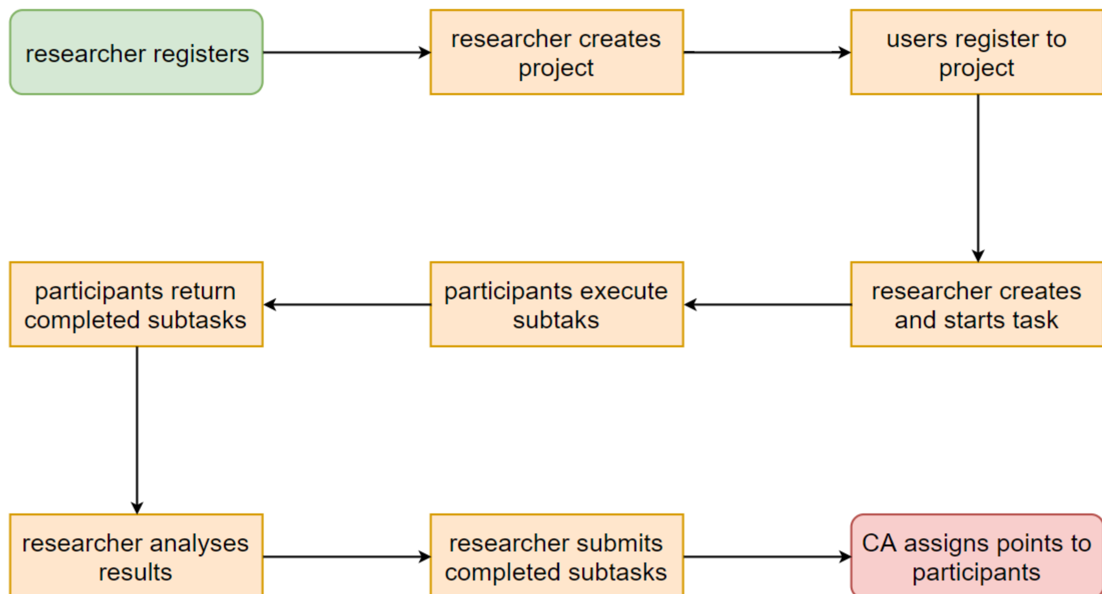


Figure 7.1: Basic procedure of the use case.

The process of the use case is depicted in Figure 7.1. The first step is the registration of the research user. This is done by sending a request to the CA using the procedure

described in Section 5.3.1. After the successful registration, the user is able to create a new *BruteForcePassword* project. The researcher uses the rules illustrated in Figure 7.2 to restrict the access to the project.

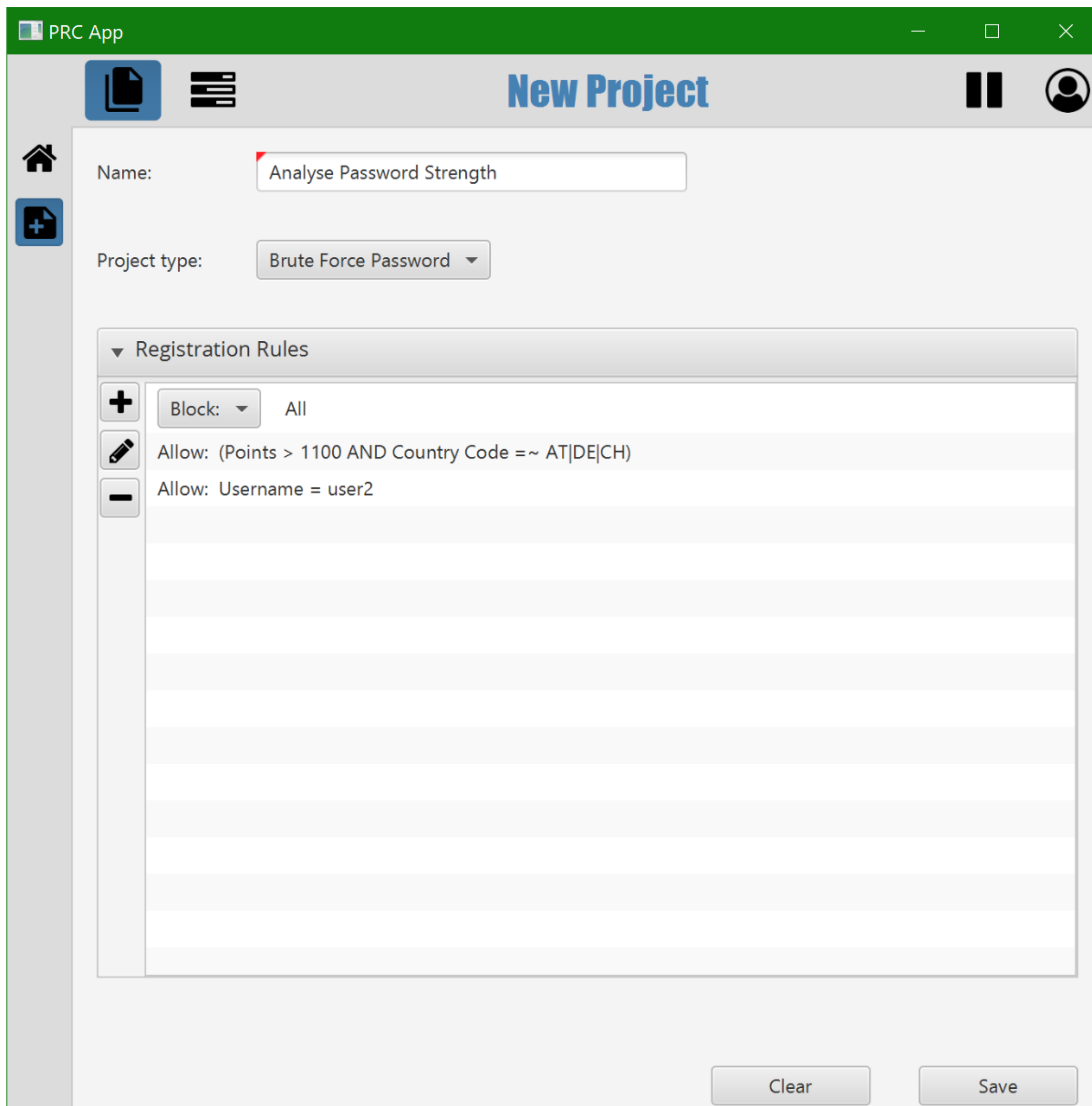


Figure 7.2: Project rules defined by the researcher.

Only users that have more than 1100 points and come either from Austria, Germany or Switzerland are allowed participating in the project. *User2*, who is a friend of the researcher, is excluded from these restrictions and is also permitted to contribute. All other users are blocked. Assuming that all previously described users try to register to the project, the following statements hold:

User1 has insufficient points and is not allowed to participate.

User2 can register to the project, because the username equals $User2$.

User3 fulfils the criteria by having 1200 points and coming from Austria.

User4 possesses enough points but lives in USA and hence cannot contribute.

It follows that only $User2$ and $User3$ pass the project rules and are able to participate in the project. As a next step, the researcher creates a task that includes an encrypted file that is secured by a password the project owner wants to test. By choosing an alphabet and the maximum length of possible passwords a list of candidates is generated and included into the task. The researcher decides to choose a redundancy factor of one to confirm the results of the users. Since $User2$ and $User3$ are registered to the project, the scientist is able to start the task. Subtasks, containing the encrypted data and a sub-list of possible passwords, are created and distributed to the participants. By producing more tasks with other passwords in the same way and measuring the time needed to find the correct one, the researcher can compare and classify the passwords by their strength. When completed subtasks arrive at the project owner, they are delegated forward to the CA server. As a result, $User2$ and $User3$ will earn points for correctly completed subtasks.

7.2.3 Fulfilled Requirements

By giving a workflow example of a researcher who wants to categorise passwords by their strength, the use case shows that the application and their functional features work as expected. As researcher and participants are able exchange subtasks with each other and do not need the help of a central server, it follows that the Peer Model offers enough expressiveness for building the given application.

7.3 Benchmark Test

The benchmark tests were executed on a desktop machine (Intel 4790K 4.0 GHz, 16GB RAM, Windows 10) using the JMH [JMH17] framework. Two functional features were tested: user registration and task distribution. The benchmarks are composed by preparation, measurement and postprocessing phases.

The first benchmark begins with the creation of a new user object that is filled with default data and a unique username. Then the system starts counting the time. The peer model is initialised, before the user registration is sent to the CA. The server accepts the request as the username does not exist in the system yet and returns signed certificates back to the user. The time measurement ends as soon as the user receives the successful registration response from the CA.

The task distribution benchmarks consists of four users, one project owner and three workers. The preparation phase consists of user registrations, project/task generation

and project subscriptions. A dummy task is used, which does not have any workload and allows for measuring the distribution that does not include any execution time of the subtasks. The generated project contains only on rule that allows all users to participate in the project. Depending on the concrete benchmark, a task will be divided into 50 for short or 250 subtasks for long runs. The measuring begins by starting the task. Subtasks are generated and transmitted to the other three users, who simultaneously process the subtasks and return them. When the project owner receives all dispatched subtasks back, the time is stopped. Afterwards all users are logged out and the state is rollbacked to the time before the task was started. This ensures the same situation for the next iteration.

Each benchmark was executed with four different setups. All peers have either access control (AC) turned on or off and use either TLS or an unencrypted communication ($TLS\ off$). Table 7.1 presents the benchmark results of user registration ($UReg$) and task distribution with 50 ($TDist50$) and 250 subtasks ($TDist250$) separated into the different configuration settings. The outcome represents the average time of 100 benchmark executions (10 forks each having five warm-ups and 10 iterations using one thread) in milliseconds. As all runtime peers operate on the same machine, the results of the benchmark test do not include any network latency.

		UReg	**TDist50**	**TDist250**
AC off	**TLS off**	1192	1096	4302
	TLS on	1212	1112	4413
AC on	**TLS off**	1220	1433	5392
	TLS on	1238	1575	5736

Table 7.1: Benchmark results in milliseconds of user registration and task distribution (50 and 250 subtasks) having access control and/or TLS enabled/disabled.

7.3.1 Interpretation of Result

By looking at the timestamps of the log file, it can be seen that the main part of the user registration process is spent in the initialisation phase of the Peer Space. This can be explained by the fact that the registration process runs only locally and that the CA server does not perform any verification of the user. The measured times for a user registration are nearly the same for all setups. It follows that the start-up time of a peer does not significantly increase by using access control and/or encrypted communication.

The benchmark results of the task distribution show that enabling access control is more expensive, especially the scenario with 250 subtasks reveals a gap of about 25%. The worker peers make use of a security rule that checks if the incoming subtask entry was sent by a project owner the user is registered to (see rule RT in Section 5.4.5). This authorisation step needs some time and has enough impact to slow secure peers down a bit.

The benchmark test shows that for the initialisation time of the Peer Space it is not

relevant if security is enabled or not. Furthermore, it can be demonstrated that a secure peer has lower performance than a peer without access control.

7.4 Comparison to the Analysed Systems from the Related Work

The related work chapter described similar public resource systems and presented a comparative table of selected criteria that are important for PRC applications. Table 7.2 shows how these attributes are fulfilled by the developed application compared to the other systems.

	distr.net	BOINC	M@Home	XWeb	F@Home	PRCApp
P2P	−	−	+	+	−	+
Scalability	∼	∼	+	+	∼	∼
Custom tasks	−	+	−	+	−	∼
AC for PO	−	−	−	−	−	+
SCC	−	+	−	+	−	+
SCE	−	+	−	+	+	+
Reward system	+	+	−	−	+	+

Table 7.2: Comparison of relevant features from analysed related systems with the application of this thesis

+: supported
∼: supported with limitations
−: not supported
SCC: Secure Communication Channel
SCE: Secure Code Execution
AC: Access Control
PO: Project Owner

Peer-to-peer network is supported as the application uses peers for communication between users, whereas there exists only one centralised CA for user registrations and point submissions. Although the users communicate in peer-to-peer network with each other, the server limits the scalability of the application. Especially the submission of subtasks has a negative impact on the server load with a growing number of users. This is the reason why scalability is only supported with limitations. A solution to this problem is presented in Chapter 8. The creation of custom tasks is marked with a tilde as users are not able to design own tasks within the application. This can only be done by extending the source code as explained in Section 6.2. Due to the fact that this modification is small and only involves the extension of two classes including the overriding of three methods, the *Custom tasks* criterion is supported with limitation. The full compatibility is planned for future work (see Chapter 8). Project owners are able to define generic policies based on user attributes. With the help of the Secure Peer

Space these rules determine which users are allowed to participate into a project. Hence, the access control attribute is satisfied. As all instances of the system talk on top of TLS, the communication channel can be seen as secure. Messages by untrusted users that are not signed by the CA are ignored by all peers in the systems. Only subtasks that were received from project owners the user is contributing in and the CA is trusting will be accepted and executed. The $PRCApp$ rewards users for sharing their resources and completing task with incentives (points).

It follows that the implemented program satisfies all criteria depicted in Table 7.2 and is the only one that supports access control.

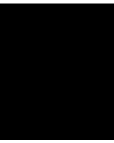
7.5 Critical Reflection

As other PRC systems, the application can be used to gain access to computational resources of volunteers. A unique feature of the program is that project owners are able to define which users are allowed to participate in their project by using attribute-based access rules. This is especially useful to protect the project against cheating users, which are detected by a similar mechanism that is used in BOINC, or to create private projects, where only chosen users are able to join.

At the beginning it was hard to understand the Peer Model, as it uses a completely different approach as other space-based middlewares. But after the initial starting difficulties the Peer Model is a powerful tool that allows designing and coordinating a peer-to-peer network in many diverse ways. Especially the modelling of the application was straight forward as peers helped to separate the features from each other and wirings have variety of designing options to choose from. A disadvantage is that the Peer Model is too complex when it comes to simple tasks. For example, it involves more effort to get all entries from a container than it would in other space-based middlewares.

The integration of the security model was very good as it further improves the expressiveness of the Peer Model by using meta-entries for peers, wirings and rules and using fine-grained access control rules. The definition of such rules for the application was intuitive and allowed to design simple rules, like granting write access to $UserPeer's$ PIC for responses of the CA server and more complex rules, e.g. regulating the access rights to a project by using self-defined policies (based on user attributes) that can be changed dynamically by the project owner. One problem that arise during the implementation of the access control rules was the lack of a documentation. Although there exist test cases and theoretical information about the Secure Peer Space, the implementation of some rules had to be done according to the trial and error principle until the desired outcome was achieved.

CHAPTER 8



Future Work

This chapter presents tasks that can be done in the future. The ideas came during the design and implementation phases.

Custom tasks would allow users to create their own project types. This would be very helpful, as the application would not be restricted any more to specific kind of tasks, but could be generally opened for executing any job that allows for distribution. However the implementation will look like, it has to be considered that letting users introduce new tasks is a big security risk, because project owners are able to execute custom code on machines of participants. Therefore, it is very important to use security mechanisms, like sandboxing or code signing, to protect other users from malicious programs.

Introducing teams (groups of users) would increase the motivation, as users can compete with each other by comparing their earned points with other team members.

Sometimes the results of subtasks needs to be further processed. For example the result of a *Sort List* subtask needs to be merged with other sorted lists to obtain the final outcome, the complete sorted list of all sorted sub-lists. This problem could be solved by introducing recursive tasks, which take the result of their subtasks, reduce them to form new subtasks and let them again be solved by the participant. This is repeated until only one subtask exists, which holds the final result.

The current implementation does not include a result for tasks, only for subtasks. It would be nice if project owners could define the outcome of a task on basis of its subtasks, e.g. the subtask which finds the right password should be the final result of the associated *Brute Force* task. As soon as the outcome of a task is found, the distribution of further subtasks can be stopped.

At the moment certificates are valid until they expire, but there exist events where it is necessary to invalidate issued certificates (e.g. stolen private key, user has malicious

intents, ...). Such a revoking mechanism could be realised by a CRL or using OCSP. Section 3.2.7 explains these two methods.

To decrease the server load, it would be possible that project owners assign the points directly to the users and only send the points to the server. This would improve the scalability of the system.

Removing old point entries on the server would save disk space and avoid memory leakage.

CHAPTER 9

Conclusion

The aim of the work was to develop a public resource application that makes use of the Secure Peer Space. Users can offer their resources by participating in projects created by other users. Project owners are able to limit the access to their projects by self-defined rules. As the Peer Model is used for communication and coordination between the individual users, the security extension allows implementing such restrictions for the owner by using ABAC. Furthermore, the application serves as a use case to confirm the expressiveness and correctness of the Secure Peer Space.

With the help of the Peer Model a P2P architecture was designed. Projects are an important part of the program, as they define the goal the owner wants to achieve and allow other users to join and provide their computational power. The workload is encapsulated into task objects that are distributed as smaller subtasks to the participants.

The Secure Peer Space uses a simple mechanism based on password lists to ensure the authenticity of users. As this approach is not applicable for the real world, a more secure solution had to be found. Digital certificates are particularly well suited for this purpose. Public key certificates are used for authentication, whereas attribute certificates assure the authorisation of a user. To ensure the integrity of communication data, TLS is integrated into the Peer Space middleware.

The program was evaluated by a use case that proves the correct behaviour of the system and illustrates the practically possible field of application. Moreover, a benchmark test that simulates task distributions points out that security mechanism of the Peer Space has performance overhead of about 25%. Important PRC criteria are analysed in a final comparison to the related systems, which showed that the program is the only one that provides access control to the project owner. The outcome of the evaluation proved that all requirements were fulfilled and that the Secure Peer Space works as intended and offers enough expressiveness to model and implement a PRC application.

List of Figures

3.1 Example of a Secure Peer Model with peers, entries, wirings and sub-peer. 16
3.2 Structure of an X.509v3 public key certificate taken from [BKW13]. 21

5.1 Internal architecture of a user. 32
5.2 User registration process with CA-server. 36
5.3 Project search and registration process. 37
5.4 Task distribution and point submission processes. 38
5.5 Structure of UserPeer and UserCAPeer including wirings and rules. 40
5.6 Structure of ProjectPeer including wirings and rules. 42
5.7 Structure of TaskPeer including wirings and rules. 45
5.8 Structure of WorkPeer including wirings and rules. 46
5.9 Structure of *PointsPeer* including wirings and rules. 48

6.1 Class diagram that gives an overview of the implementation. 52
6.2 Class diagram with tasks and subtasks. 53
6.3 An overview of the Secure Peer Space implementation including adaptations. 54
6.4 Screenshot of the registration form. 62
6.5 Screenshot of the *Project Home* view. 63
6.6 Screenshot of creating a new project. 64
6.7 Screenshot of generating a new brute force task. 65
6.8 Screenshot of the *Contribution Home* view. 66
6.9 Screenshot of the *User Menu*. 67
6.10 Screenshot of the point statistics view. 67
6.11 Screenshot of changing user attributes. 68

7.1 Basic procedure of the use case. 72
7.2 Project rules defined by the researcher. 73

List of Tables

2.1 Comparison of relevant features from analysed related systems 11

7.1 Benchmark results in milliseconds of user registration and task distribution (50 and 250 subtasks) having access control and/or TLS enabled/disabled. 75

7.2 Comparison of relevant features from analysed related systems with the application of this thesis 76

Listings

6.1 Server.init() method in peer communication package 57
6.2 Wiring that submits subtasks to CA 59
6.3 Rule that allows to write and take wirings from *ProjectPeer* 59
6.4 Project rule (R_2) from Section 5.4.3 60
6.5 Entry predicate of the project rule 60

Acronyms

AA attribute authority. 22, 23

ABAC attribute-based access control. 21, 27, 81

AC attribute certificate. 21–23, 31, 34–36, 40, 41, 56, 57, 62, 67, 68

AES Advanced Encryption Standard. 18, 61

API Application Programming Interface. 6, 11, 24

ASN.1 abstract syntax notation version 1. 19–21

BOINC Berkeley Open Infrastructure for Network Computing. 6, 7, 77

BSE Bovine spongiform encephalopathy. 10

CA certificate authority. 17, 22, 23, 31, 34–36, 38–41, 45–47, 56–58, 61, 63, 66–68, 70–72, 74–77

CFIM Closed Frequent Itemsets Mining problem. 8

CLI command line interface. 51

CRL certificate revocation list. 23, 24, 80

CSP Certificate Service Provider. 22, 23

CSR certificate signing request. 23, 35, 39–41

DDOS distributed denial-of-service. 10

DER distinguished encoding rules. 19–21

DES Data Encryption Standard. 18

DEST destination. 13, 14, 17, 58

DN distinguished naming. 19–21, 56

EC entry collection. 14, 16

GPU Graphics Processing Unit. 7

GUI graphical user interface. 24, 29, 32, 33, 40, 51, 54, 58, 60, 61, 69, 70

HSM hardware security module. 23, 29

HTTP Hypertext Transfer Protocol. 58

HTTPS Hypertext Transfer Protocol Secure. 7, 11

IDE Integrated Development Environment. 25

IETF Internet Engineering Task Force. 19

ITU-T International Telecommunication Union Telecommunication Standardization Sector. 19, 22

LIGO Laser Interferometry Gravitational Observatory. 7

OCSP online certificate status protocol. 23, 24, 80

OID object identifier. 19–21

PIC peer-in-container. 1, 13, 14, 16, 17, 39–41, 43–45, 47, 48, 58, 61, 77

PKC public key certificate. 20–22, 31, 34–36, 40, 41, 56, 57, 68

PKI Public Key Infrastructure. 3, 13, 17, 18, 24, 55

POC peer-out-container. 1, 13, 16, 44, 46, 49, 58

POM project object model. 25

PRC Public Resource Computing. 1–3, 5, 6, 10, 27, 31, 61, 76, 77, 81

PSC Peer Specification Container. 16, 17

RA registration authority. 22, 23

SPC Security Policy Container. 16, 17, 47

SSL Secure Sockets Layer. 9, 11, 24

TLS Transport Layer Security. 17, 22, 24, 31, 34, 35, 55–58, 70, 71, 75, 77, 81, 85

TTL time-to-live. 13

TTS time-to-start. 13

URI Uniform Resource Identifier. 13

URL Uniform Resource Locator. 7

VA validation authority. 23

WSC Wiring Specification Container. 16, 17, 42–44, 47–49

Bibliography

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[All16] Bruce Allen. Einstein@home homepage. `https://einsteinathome.org`, 2016. [Online; accessed 2016-11-28].

[And04] David P. Anderson. BOINC: A system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10. Institute of Electrical and Electronics Engineers (IEEE), 2004.

[And07] David P. Anderson. Creating BOINC projects. `https://boinc.berkeley.edu/boinc.pdf`, 2007. [Online; accessed 2018-02-20].

[And16a] David P. Anderson. BOINC homepage. `https://boinc.berkeley.edu/`, 2016. [Online; accessed 2016-11-28].

[And16b] David P. Anderson. BOINC security homepage. `https://boinc.berkeley.edu/wiki/BOINC_Security`, 2016. [Online; accessed 2016-11-28].

[Apa17] Apache maven project. `https://maven.apache.org/`, 2017. [Online; accessed 2017-08-21].

[BEJ+09] Adam L. Beberg, Daniel L. Ensign, Guha Jayachandran, Siraj Khaliq, and Vijay S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[BKW13] Johannes A. Buchmann, Evangelos Karatsiolis, and Alexander Wiesmaier. *Introduction to Public Key Infrastructures*. Springer Publishing Company, Incorparated, 2013.

[Bra08] Daren C. Brabham. Crowdsourcing as a model for problem solving: An introduction and cases. *Convergence*, 14(1):75–90, 2008.

[CDF+05] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frédéric Magniette, Vincent Néri, and Oleg Lodygensky. Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. *Future generation computer systems*, 21(3):417–437, 2005.

[CDJ+13] Stefan Craß, Tobias Dönz, Gerson Joskowicz, Eva Kühn, and Alexander Marek. Securing a space-based service architecture with coordination-driven access control. *JoWUA*, 4(1):76–97, 2013.

[CJK15] Stefan Craß, Gerson Joskowicz, and Eva Kühn. A decentralized access control model for dynamic collaboration of autonomous peers. In *International Conference on Security and Privacy in Communication Systems*, pages 519–537. Springer, 2015.

[De 11] De Soete, Marijke. Attribute certificate. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 51–52. Springer US, Boston, MA, 2011.

[Des16] Travis Desell. MilkyWay@home homepage. `http://milkyway.cs.rpi.edu/milkyway/`, 2016. [Online; accessed 2016-11-28].

[dF16] Gianni de Fabritiis. GPUGrid@home homepage. `https://www.gpugrid.net/`, 2016. [Online; accessed 2016-11-28].

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[Dis16] Distributed.net homepage. `https://www.distributed.net/`, 2016. [Online; accessed 2017-07-14].

[DR01] Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 75–80. IEEE, 2001.

[DR08] Tim Dierks and Eric Rescorla. Transport layer security (TLS) protocol version 1.2. *IEFT RFC 5246*, 2008.

[Ecl17] Eclipse IDE. `https://eclipse.org/`, 2017. [Online; accessed 2017-08-21].

[FKK11] Alan Freier, Philip Karlton, and Paul Kocher. Secure sockets layer (SSL) protocol version 3.0. *IEFT RFC 6101*, 2011.

[Gel85] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[Gil98] John Gilmore. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design.* O'Reilly & Associates, Incorporated, 1998.

[IET18] IETF internet engineering task force. https://www.ietf.org/, 2018. [Online; accessed 2018-02-20].

[ITU18] ITU telecommunication standardization sector. https://www.itu.int/en/ITU-T/, 2018. [Online; accessed 2018-02-20].

[Jav17a] Java homepage. https://www.java.com/, 2017. [Online; accessed 2017-08-21].

[Jav17b] JavaFX home. https://docs.oracle.com/javafx/, 2017. [Online; accessed 2017-08-21].

[JMH17] Java microbenchmark harness (JMH) homepage. http://openjdk.java.net/projects/code-tools/jmh/, 2017. [Online; accessed 2017-08-21].

[KB05] Yoram Kulbak and Danny Bickson. The emule protocol specification. *The Hebrew University of Jerusalem, School of Computer Science and Engineering*, 2005.

[KCJ+13] Eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-based programming model for coordination patterns. In *International Conference on Coordination Languages and Models*, pages 121–135. Springer, 2013.

[KMKS09] Eva Kühn, Richard Mordinyi, László Keszthelyi, and Christian Schreiber. Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 625–632. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[Kor16] Eric Korpela. SETI@home homepage. https://setiathome.berkeley.edu/, 2016. [Online; accessed 2016-11-28].

[KPP+06] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA – a cost-optimized parallel code breaker. In *Proceedings of the 8th international conference on Cryptographic Hardware and Embedded Systems*, pages 101–118. Springer-Verlag, 2006.

[LMOT10] Claudio Lucchese, Carlo Mastroianni, Salvatore Orlando, and Domenico Talia. Mining@home: toward a public-resource computing framework for distributed data mining. *Concurrency and Computation: Practice and Experience*, 22(5):658–682, 2010.

[MKL+02] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing. *HPL-2002-57 (R.1)*, 2002.

[MSKH02] James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*, 2002.

[MTV05] Carlo Mastroianni, Domenico Talia, and Oreste Verta. A super-peer model for resource discovery services in large-scale grids. *Future Generation Computer Systems*, 21(8):1235–1248, 2005.

[Nyk00] Toni Nykänen. Attribute certificates in X.509. *Helsinki University of Technology, Department of Computer Science and Engineering*, 2000.

[Ora17] Oracle JDK 8 and JRE 8 certified system configurations. `http://www.oracle.com/technetwork/java/javase/certconfig-2095354.html`, 2017. [Online; accessed 2017-07-19].

[Pan13] Vijay S. Pande. Folding@home homepage. `https://folding.stanford.edu/`, 2013. [Online; accessed 2017-02-06].

[Ram99] Blake Ramsdell. S/MIME version 3 message specification. *IEFT RFC 2633*, 1999.

[Rex04] Blerim Rexha. Securing web services in a user-to-application model based on certificate private extensions and smartcard technology. Master's thesis, TU Wien, 2004.

[Rie12] Christian B. Ries. *BOINC - Hochleistungsrechnen mit Berkeley Open Infrastructure for Networking Computing*. Springer Publishing Company, Incorporated, 2012.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Spi09] Terence Spies. *Computer and Information Security Handbook*. Elsevier, Incorparated, 2009.

[U.S99] U.S. Department of Commerce/National Institute of Standards and Technology. Data encryption standard (DES). *Federal Information Processing Standards (FIPS) Publication*, 46-3, 1999.

[U.S01] U.S. Department of Commerce/National Institute of Standards and Technology. Advanced encryption standard (AES). *Federal Information Processing Standards (FIPS) Publication*, 197, 2001.

[WK17] George Woltman and Scott Kurowski. GIMPS homepage. `https://www.mersenne.org/`, 2017. [Online; accessed 2017-12-22].

[YHK93] W. Yeong, T. Howes, and S. Kille. X. 500 lightweight directory access protocol. Technical report, ITU-T RFC 1487, 1993.