

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



Administrative Profiles Unit and Multi-Sessions Management Scheme for IPBrick Private Cloud

Anwaar Hussain

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Internal Supervisor: Prof. João Neves

External Supervisor: Engr. Miguel Ramalhão

External Co-Supervisor: Mr. Helder Santos

July 29, 2016

Abstract

IPBrick solution in the form of a Private Cloud has been well-received by its customers since its launch in the market. However, it has its own limitations in spite of having some distinct attributes. As a result, it can be challenged by its alternatives in the market present at the moment. Currently, the Private Cloud IPBrick is operated by a single user who manages every aspect of it by using the given set of privileges in his/her profile. With the growing demand of IPBrick solution in a circle of some mega-sized firms, it has become difficult for one administrator to handle the management load. Thus, there is a need of distributing the cloud's access among the multiple administrators by allocating them a profile of their own which will simplify the administration process significantly. Other prominent limitation of IPBrick SaaS based solution is its less effectiveness in an environment driven by the multi-sessions due to which any level of data conflict in the database can not be mitigated during the concurrent execution of the two or more transactions. For this purpose, a procedure should be implemented in the IPBrick system in-order to preserve the integrity and the consistency of data stored in the database.

This thesis provides the detailed description of implementations that had been carried out for the modules: Administrator Profiles Unit and Multi-Sessions Management Scheme. It also contains the literature review which had been covered in the beginning to make sure that the development phase could be conducted in a best possible fashion. The conceptualization stage along with the applied techniques has been discussed in this thesis that led to the successful accomplishment of all objectives. At the end, some limiting factors are also mentioned due to which the efficiency of both features got constrained to some extent.

Acknowledgements

Firstly, I would like to thank my thesis advisor Prof. João Neves of the Department of Electrical and Computer Engineering at Faculty of Engineering, University of Porto. I had an affiliation with Prof. Neves as a student even before performing my dissertation under his supervision and I always found him truly inspirational. Throughout this thesis, he had allowed me to work on my ideas and steered me in the right direction whenever it was required.

I would also like to thank my supervisor at IPBrick Engr. Miguel Ramalhão for trusting in my abilities and letting me work on this thesis. His intuitive inputs were really useful in bringing a more pragmatic approach to my research work. I am also grateful to co-supervisor Mr. Helder Santos at IPBrick for his technical assistance during the implementation phase of the dissertation.

Finally, I must express my very profound gratitude to my family – especially, my father Engr. Ashiq Hussain and my brother Dr. Bilal Hussain – and friends for supporting me relentlessly throughout my years of studies. The accomplishment of writing this thesis would not have been possible without their presence around me. Thank you all!

Anwaar Hussain

*“Logic will get you from A to B.
Imagination will take you everywhere.”*

Albert Einstein

Contents

1	Introduction	1
1.1	Background Context	1
1.2	Objectives	2
1.3	Contribution	3
1.4	Structure of the Thesis Report	4
2	Literature Review	5
2.1	Web Languages and Technology	5
2.1.1	Static Web Pages	5
2.1.2	Dynamic Web Pages	6
2.1.3	Comparison Between Static and Dynamic Pages	8
2.2	Concurrency Control Mechanisms in Databases	8
2.2.1	Pessimistic Concurrency Control	11
2.2.2	Optimistic Concurrency Control	12
2.2.3	No check Concurrency Control	13
2.2.4	Hybrid Concurrency Control	13
2.2.5	Comparison between Optimistic and Pessimistic Approach	14
2.3	Conclusion	15
3	Refinement of IPBrick Solution	17
3.1	Introduction	17
3.2	Administrative Profiles Unit	17
3.2.1	Process of Profile Insertion	18
3.2.2	Process of Profile Modification	19
3.2.3	Process of Profile Deletion	20
3.2.4	Process of Profile Allocation	20
3.2.5	Process of Profile De-Allocation	21
3.2.6	Process of Profile View	21
3.3	Multi-Sessions Management	22
3.3.1	Implications in Concurrency Control at Low Level	22
3.3.2	Solution in the form of Concurrency Control at High Level	23
3.4	Conclusion	24
4	Implementation of Modules and Results	25
4.1	Introduction	25
4.2	IPBrick System	25
4.2.1	Database	25
4.2.2	Web Interface	26

4.3	Implementation of Administrative Profiles Unit	27
4.3.1	Inclusion of new Tables	27
4.3.2	Inclusion of new Web Pages	30
4.3.3	Modification of existing Web Pages	38
4.3.4	Inventory of new Web Pages in database	43
4.4	Implementation of Multi-Sessions Management	44
4.4.1	Table <i>sessao</i>	44
4.4.2	Table <i>alteracao</i>	44
4.4.3	Table <i>transaction</i>	45
4.4.4	Scheme as a Final Solution	46
4.5	Extra Work	48
4.5.1	Renaming of Web Pages' Names in the Database	48
4.5.2	Run-Time Configurations	49
4.6	Results	53
4.6.1	Performance Analysis of Administrative Profiles Unit	53
4.6.2	Performance Analysis of Multi-Sessions Management Scheme	55
4.7	Conclusion	56
5	Conclusion and Future Work	57
5.1	Accomplished Goals	57
5.2	Future Work	58
A	Useful Code Snippets	61
A.1	Functions for Profile Insertion	61
A.2	Function for the specification of Profile Permissions	62
A.3	Funtions for Profile's Allocation	68
A.4	Funtions for applying permission on a web page	69
B	Relational Algebra	71
B.1	Administrative Profiles Unit	71
B.1.1	Queries for the <i>profile</i> Table	71
B.1.2	Queries for the <i>profile_menu</i> Table	71
B.1.3	Queries for the <i>profile_page</i> Table	72
B.2	Multi-Sessions Management Unit	72
B.2.1	Queries for the <i>transaction</i> Table	72
B.2.2	Queries for the <i>sessao</i> Table	73
C	Mock-ups of web pages	75
D	Utility Stuff	79
D.1	Bash commands	79
D.2	PHP Functions	79
D.3	Structure of the SOAP Request/Response Message	79
D.4	Structure of WSDL	80
	References	83

List of Figures

2.1	Typical Architecture of a Static Website	6
2.2	Typical Architecture of a Dynamic Website	7
2.3	Centralized Database System	11
3.1	Flow chart of the Profile Insertion Process	18
3.2	Flow chart of the Profile Modification Process	19
3.3	Flow chart of the Profile Deletion Process	20
3.4	Flow chart of the Profile Allocation Process	21
3.5	Flow chart of the Process handling Multiple Transactions	24
4.1	Index web page of IPBrick Web Interface	26
4.2	<i>profile</i> Table along with its fields and entries	28
4.3	<i>profile_menu</i> Table along with its fields and rows	29
4.4	<i>profile_page</i> Table along with its fields and rows	29
4.5	Entries in <i>valida</i> Table along with a new addition of <i>idprofile</i> column	30
4.6	Web Page with a prompt for the Profile Definition	32
4.7	Web Page with the Collapsed view of Profile's Permissions	32
4.8	Web Page with View of Profile's Permissions	34
4.9	Web Page for the modification of an existing Profile	35
4.10	Web Page for the deletion of a Profile	35
4.11	Web Page for the selection of a profile from the list	36
4.12	Web Page for the allocation of a user to an existing Profile	37
4.13	Web Page for the removal of a user from accessing an existing Profile	37
4.14	Page for viewing an existing Profile	38
4.15	Web Page displaying the access details	39
4.16	Possible access to the default web page before the modifications in <i>corpo</i> script	40
4.17	No access to the default web page after the modifications in <i>corpo</i> script	41
4.18	Default web page with All Menus and Sub Menus	42
4.19	Default web page with Few Menus and Sub Menus	42
4.20	Table <i>pages</i> entries along with the definition of its fields	43
4.21	Table <i>links</i> entries along with the definition of its fields	43
4.22	Table <i>liga_pages_links</i> entries along with the definition of its fields	44
4.23	Table <i>sessao</i> rows along with the definition of its fields	45
4.24	Table <i>alteracao</i> rows along with the definition of its fields	45
4.25	Table <i>transaction</i> entries for one session along with the definition of its attributes	46
4.26	Table <i>transaction</i> entries for two sessions along with the definition of its attributes	47
4.27	An alert for the second Admin if he wants to modify the same data as the first one	47
4.28	Table <i>pages</i> containing pages' names based on a new nomenclature in <i>page</i> column	49

4.29 Sitemap of Administrative Profiles Unit	53
C.1 Mock-up of the default web page of Administrative Profiles Unit	75
C.2 Mock-up of the web page for Profile Insertion	76
C.3 Mock-up of the web page for the selection of Profile's Permissions	76
C.4 Mock-up of the web page for the modification in the Profile's Definitions	77
C.5 Mock-up of the web page for the modification in the Profile's Permissions	77

List of Tables

2.1	Pros and Cons of Static and Dynamic Web Pages	8
2.2	Example of Pessimistic Concurrency Control	12
2.3	Example of Optimistic Concurrency Control	13
3.1	Pros and Cons of Optimistic and Pessimistic Concurrency Controls	23
4.1	Facts about the databases <i>Systemsoft</i> and <i>Systemconf</i>	26
4.2	Privileges of Administrators having Profiles of the different types	40
4.3	Concurrency Control between N transactions related to the same entity's data . .	48
4.4	Parameters of the function <i>applyAddUserRuntime</i>	50
4.5	Data Set of Test Scenario for Administrative Profiles Unit	54
4.6	Execution Time of the different Operations in the Administrative Profiles Unit . .	54
4.7	Data Set of Test Scenario for Multi-Sessions Management Scheme	55
4.8	Execution Time of the different Operations in the Multi-Sessions Management Scheme	56

List of Abbreviations/Acronyms

2PL	Two Phase Locking
Admin	Administrator
BOCC	Backward Oriented Concurrency Control
CMS	Content Management System
CSS	Cascading Style Sheets
DM	Database Manager
DBMS	Database Management System
DBS	Database System
DHTML	Dynamic HyperText Markup Language
FOCC	Forward Oriented Concurrency Control
HTML	HyperText Markup Language
PHP	HyperText Preprocessor
Postgres	PostgreSQL
RPC	Remote Procedure Calls
SaaS	Software as a Service
SOAP	Simple Object Access Protocol
WSDL	Web Services Description Language
XHTML	Extensible Hyper-Text Markup Language

Chapter 1

Introduction

This chapter covers the background context of the given topic comprehensively to make the objectives of the thesis more understandable for the reader. It also exhibits the author's motivation behind choosing this thesis' theme and his sincere intentions toward producing a solid contribution in the field of Engineering along with some productive enhancements in IPBrick solution.

1.1 Background Context

“Science never solves a problem without creating ten more.”

George Bernard Shaw

Since the advent of the concept SaaS in the field of Telecommunication Engineering, IPBrick has been one of few well-known firms to develop an efficient solution based on SaaS for the unified Communication Services and it has received an overwhelming response right from the very beginning [1]. IPBrick solution is quite unique because it does not require any special hardware as the slogan of IPBrick says "XXI century Software in a XIX century Hardware". In spite of having some distinctive attributes, it is never easy for any product of high quality pertaining to the ever-changing sector of Information Technology to envelope all of its paradigm shifts effectively. Such peculiar is the case of IPBrick solution in the form of Private Cloud that needs to overcome its limitations on an immediate basis to maintain its reputation among the customers. From the beginning till date, the administrative access to IPBrick Private Cloud is limited to only admin who carried all privileges because when the idea of this solution was conceived, no one thought that it will have a high demand among the customers base on some day. Today, IPBrick solution has got a very strong word of mouth in the market, many big companies and firms are interested in buying it with some definite enhancements. One of the improvements is an expansion in the range of its administrative access from one admin to multiple admins which in other words means that IPBrick solution should allow distributing a set of profiles among the multiple admins; so that they can practice the privileges defined in their respective Profiles in the cloud to make its

management process lot easier. IPBrick solution was under-estimated on one other front too and that is its development for a single-session based mode of operation persistently. It had tackled the challenges of the last decade but now it needs to be improved further and should be made compatible with the multi-sessions based operations. Here, it should be noted that IPBrick clients from the Hospital São João, Porto have already requested for this feature; but it's not possible in the current scenario because the existing solution is not fully ready for it. In an environment driven by the multi-sessions, the consistency and the integrity of clients' data stored in the database should not be taken for granted. For this purpose, it is essential to develop a mitigation scheme which can avoid any sort of data conflicts in the database without affecting much the operations during the execution of concurrent transactions. So, it's high time for IPBrick to address the issues mentioned in this section in-order to prolong the viability of its solution

1.2 Objectives

“There is no achievement without goals.”

Robert J. McKaine

The main aim of this thesis is to improve IPBrick Private Cloud based solution by simplifying its management process and also by making it highly compatible with the setup based on multi-sessions. Those improvements can be achieved by the accomplishment of two main objectives which are following:

- Develop a Profiles Management Unit that will allow the multiple admins to perform their managerial operations on the IPBrick private cloud by using the set of permissions offered in their respective profiles. Any admin with an access to IPBrick administrator will be able to visit only those Web Pages and Menus/Sub-Menus which will lie inside the domain of his/her profile and will not be permitted to perform any operation (Insert/Delete/Modify/All) on any service other than the specified one. A profile will only be created by an administrator assigned with such privileges and it will be available for the modification and deletion operations for the same admin. Those operations on a profile should also be applicable by any other admin with the same or higher level of permissions as its creator¹.
- Implement a multi-sessions management ² scheme which will ensure the concurrency control during the parallel execution of the multiple transactions. This scheme will mainly deal with the Update or Delete operations performed on the data stored in IPBrick database while the other operations such as Write and Read will be executed without any interruption/delay. Not all of the concurrently generated transactions by the different admins will result into the

¹An admin with customized set of privileges can also create, edit and delete some profile but that permission will be subject to the definition of his/her profile

²In this thesis, multi-sessions management module only deals with the concurrency control in database accessed by the multiple admins simultaneously.

modification/deletion of the same entity's data³. But if by any chance that possibility occurs then the transaction with the first access to an entity's data will be permitted to perform an operation on it while rest of other transactions will be rejected and notified accordingly to their respective admins.

1.3 Contribution

“It is the greatest of all mistakes to do nothing because you can only do little - do what you can.”

Sydney Smith

This thesis contributes to the area of web services and databases in quite a pragmatic way. It highlights two main points: Significance of the multiple administrators in a private cloud and the assurance of data consistency in a multi-sessions based environment, There are many firms like IPBrick all around the world providing their set of services in the form of a cloud. It is easy to handle the administrative operations when the domain of a cloud is limited in size. Otherwise, it gets cumbersome for one or two administrators to deal with all of the management related tasks and such extra-ordinary situations demand for some extra-ordinary measures. IPBrick R&D team has foreseen the managerial issues in their SaaS based solution which will appear more prominently soon due to the growing range of services in IPBrick Cloud. That's why Administrative Profiles Unit was made part of this thesis to address many upcoming issues in advance. This module is not only specific to IPBrick solution, can be used in any Cloud based solution because of the simplistic approach that has been used for its implementation while keeping user's ease of use in view. The second objective of this thesis provides a mechanism through which the data's consistency and integrity can be assured in a setup handled by multi-sessions. Without its implementation, the adverse effects are beyond one's imagination especially when two or more users are using the same program in their respective sessions. In a way, this particular goal has more significance than the other one because it deals with the concept of concurrency control in database which is usually discussed theoretically in the Databases' courses without any intent of its actual implementation in the real/non-realtime applications. This thesis provides a different approach that is to implement the concurrency control mechanism at the abstract level. The conventional Concurrency Control at the low level(Database Layer) was not easily applicable in the IPBrick database due to its large size and formation. Within the given time frame, another approach was devised for the multi-sessions management that has its limitations but still quite effective enough to avoid some level of data conflicts in the database without any requirement of some intensive implementations. It is hoped that the methodology for concurrency control discussed in this thesis will become a reference point for some other research works dealing with the similar constraints and scenarios.

³A data belongs to a user or service of IPBrick.

1.4 Structure of the Thesis Report

Besides the Introduction, this thesis contains 4 more chapters. The Chapter 2 presents the literature covered relevant to the given topic. The devised solution for each objective of this thesis along with its limitations is discussed in the Chapter 3. The Chapter 4 explains the actual implementations that had been performed along with the critical analysis of their results obtained afterwards. Lastly, the Chapter 5 is the final conclusion of this thesis followed by some possible future enhancements in its work.

Chapter 2

Literature Review

This chapter describes the modern day practices relevant to the goals of thesis that were specified in the previous chapter. The Section 1 of this chapter presents the mechanisms used in the development of web pages along with the involved languages such as: HTML, PHP, JavaScript and jQuery. The Section 2 of this chapter exposes a set of mechanisms relevant to the development of concurrency control scheme in database for avoiding data conflicts in it.

2.1 Web Languages and Technology

The web pages are filled with information through two mechanisms: statically and dynamically. It is important to understand each of these mechanisms deeply in-order to use them according to the requirements. And also each of those types of web pages require different set of skills without which their implementation can not be carried out in a best possible way.

2.1.1 Static Web Pages

A static web page (sometimes called a open/flat page/stationary page) is a web page that is delivered to the user exactly as stored, in contrast to the web pages which are generated by a web application. HTML is the first tool used to develop web pages. It is a language to stylize the text, create paragraphs and line breaks etc. But the most important aspect of HTML is its link creation ability between the different documents and which are made by using the HTML tags [2, 3]. Web pages created only with HTML are static pages because they remain same for all time to come until and unless these are changed manually. The retrieval of web pages from web server is done through Request and Response technique as shown in the Figure 2.1 [4]. During the static mode of operation, all of the code (HTML, CSS and Javascript) and other resources are downloaded in one go. Static web pages are suitable for the contents that never or rarely need to be updated. However, maintaining large numbers of static pages as files can be impractical without automated tools.

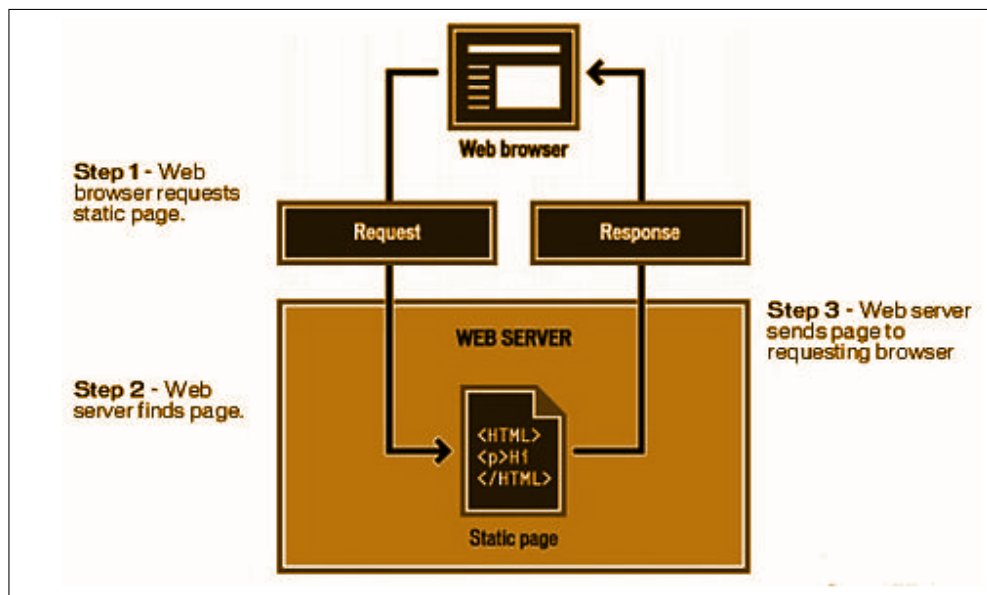


Figure 2.1: Typical Architecture of a Static Website

In-order to add few interactive features in static web pages, some scripting languages like JavaScript and JQuery can be used. These languages can add some dynamic elements in static web pages. For example, with JavaScript it is possible to display a random number every time a web page is reloaded in the browser, such pages are also called static pages. JavaScript is a prototype-based, dynamic, object-oriented, imperative and functional language [5]. It is most commonly used in web pages as a client-side scripting language¹. To make the use of JavaScript easier on web pages, jQuery can also be used. It is a lightweight, "write less, do more", JavaScript library [6]. jQuery takes a lot of common tasks that require many lines of JavaScript code to accomplish, and wraps them into methods that can be called with a single line of code. As a result, HTML document's traversal and manipulation become simple and more manageable. jQuery also provides some of its applications in the event handling and animations [7].

2.1.2 Dynamic Web Pages

As the need of creating the interactive web pages arose, the relevant means were developed to handle the demand. For instance, the product list in an on-line shopping website will be shown by the web pages with dynamic implementation. Such dynamic pages have two main components: Style of the web page and Information carried by the web page. As it is known that the style of web page is created by using the languages like HTML, CSS and JavaScript etc but they can not retrieve the content dynamically from the database. A dynamic mode of operation for web

¹Languages such as DHTML, XHTML etc that are executed at Client-Side by the user's web browser instead of web Server are called Client-Side Scripting Languages

pages can be observed in the Figure 2.2 [4]². When a web page loads in the browser, it makes a request to the linked database for furnishing some information depending upon the user's input(s). In response, it receives the required data from the database and displays the resulting web page in the browser after applying the styling on it. For example, on an online shopping website, if the user wants to see only the mobile phones available in the store, the connected database will only send the required information related to the cell phones.

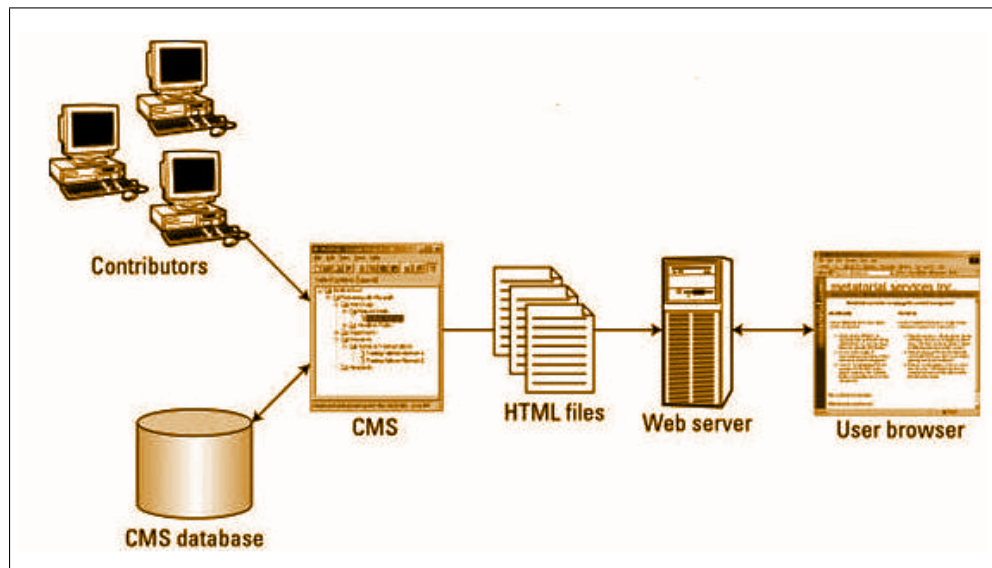


Figure 2.2: Typical Architecture of a Dynamic Website

Due to the inclusion of database in web architecture, a long list of technologies that can be used to make the web pages more dynamic in nature. Some of the important languages based on Server-Side Scripts³ that can interact with database and build the web pages at backend are: PHP, ASP, ASP.net, JSP, Perl, Python, etc. PHP was originally known as Personal Home Page and now, it is known as "Hypertext Pre-Processor" which is a recursive acronym [8, 9]. It is a dynamically typed programming language and usually used to create dynamic web pages but can also be used to create standalone programs. Besides interfacing with a database, PHP can perform all types of mathematical and scientific calculations for example: figuring out what day it is or what day of the week March 18, 2046 will be, performing all different types of mathematical equations and many others. It can also collect the user's information by letting the user to interact with the developed script directly. PHP provides many applications pertaining to graphics that allow creating simple graphics on the fly [10].

²CMS is a computer application that supports the creation and modification of digital content using a common user interface and thus usually supporting multiple users - also known as Contributors - working in a collaborative environment.

³Scripts on a web server which produce a customized response for each user/client's request to the website are called Server-Side Scripts.

2.1.3 Comparison Between Static and Dynamic Pages

A brief comparison between Static and Dynamic Web Page in terms of their Pros and Cons is listed in the Table 3.1.

Table 2.1: Pros and Cons of Static and Dynamic Web Pages

<i>Web Page Architecture</i>	<i>Pros</i>	<i>Cons</i>
<i>Static</i>	<ul style="list-style-type: none"> • Quick to develop • Cheap to develop • Cheap to host 	<ul style="list-style-type: none"> • Successful without resulting into data conflict • Much easier to update when a data conflict occurs • New content brings users back to the site and helps in the search engines • Can work as a system to allow staff or users to collaborate
<i>Dynamic</i>	<ul style="list-style-type: none"> • Provides a wide range of Interactive Features • Content can get stagnant 	<ul style="list-style-type: none"> • Slower / more expensive to develop • Hosting costs a little more • Requires web development expertise to update site

2.2 Concurrency Control Mechanisms in Databases

A database consists of a set of named data items. Each data item has a value. The values of the data items at any one time comprise the state of the database [11]. In practice, a data item could be a word of main memory, a page of a disk, a record of a file, or a field of a record. The size of the data contained in a data item is called the granularity of the data item [11]. In this work, it is assumed that data items are atomic i.e. a whole data item is accessed as one unit.

A DBS is a collection of hardware and software modules that support commands to access the database, called database operations (or simply operations) [11]. The most important operations are Read and Write. Read[x] returns the value stored in data item x without changing the state of x. Write[x] changes the value of x by overwriting the old value. Other operations have their own significance and will be brought under use from time to time.

The DBS⁴ executes each operation atomically which means that the DBS behaves as it is executing operations sequentially i.e. one at a time [12]. The DBS also supports transaction operations: begin, commit, and abort. A transaction program reports the DBS that it is about to begin executing a new transaction by issuing the operation begin. It marks the termination of the transaction by issuing either the operation commit or the operation abort. By issuing a commit, the transaction reports to the DBS that the transaction has terminated normally and all of

⁴The abbreviations DBS and DBMS are used interchangeably in this thesis.

its effects should be permanent but abort shows the abnormal termination of the transaction and its effects should be destroyed. After the DBS executes a transaction's commit /abort operation the transaction is called committed/aborted [11]. A transaction that has issued its begin operation but is not yet committed or aborted is called active. A transaction is uncommitted if it is aborted or active. If the transaction is aborted, it can be restarted. It is assumed that each transaction is self-contained, meaning that it performs its computation without any direct communication with other transactions or users [13].

A major aim of developing a database system is to allow the several users to access the shared data concurrently [13]. Concurrent access is easy if all users are only reading data because there is no way for them to interfere with one another. However, when two or more users are accessing the database concurrently and at least one is updating data, there may be interference that can cause inconsistencies [14]. Although two transactions may be correct in themselves, the interleaving operations may produce an incorrect result, thus risking the integrity and consistency of the database [13, 14, 15]. The ACID properties of a transaction that all transactions should have are [15]:

- **Atomicity** — A transaction's changes to the state are atoms: either all happen or none happen.
- **Consistency** — A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. The formal transaction maintains the consistency.
- **Isolation** — Even though transactions execute concurrently, it appears to each transaction T that other executed either before T or after T but not both.
- **Durability** — Results of the committed transaction are persistent until another committed transaction possibly changes them.

The aim of concurrency control methods is to schedule transactions in such a way as to avoid any interference [13]. One obvious solution would be to allow only one transaction to execute at a time. However, the goal of a multi-user database system is also to maximize the degree of concurrency or parallelism in the system, so that transactions can execute without interfering with one another and can run simultaneously [14]. When two or more transactions executes concurrently, their database operation execute in an interleaved way [13]. Therefore, operations from one transaction may execute between two operations from another transaction. This interleaving can cause transactions to behave incorrectly. Hence, an interleaved transaction execution can lead to an inconsistent database state. To avoid this and other problems the interleaving between transactions must be controlled [14]. The order of some operations is of no importance for the final result of executing the transactions, for example, it is possible to interchange the order of any two read operations without changing the behavior of the transaction(s) doing the reads but it can result in conflict in another case. It is said that a conflict between two transactions occurs when two

transactions operate on the same data item and at least one of the operations is write. Execution interleaving can be modeled using a history(a prefix of complete history).

One method to avoid interference problems is not to allow transactions to be interleaved at all. An execution in which no two transactions are interleaved is called serial [13]. A serial history represents an execution in which there is no interleaving of operations of different transactions. Execution is serializable [13] if it produces the same output and has the same effect on the database state as some serial execution of the same transactions. Because serial executions are correct and each serializable execution has the same effect as a serial execution that is correct too [14]. Serialization isolation can force applications to repeat a lot of work if a big transaction aborts or if some conflicts occur. It's very useful for complex cases where attempting to use row locking might just cause deadlocks, though [16]. To ensure correctness in the presence of failures, the execution of transactions should not be only serializable but also recoverable, avoid cascading aborts, and be strict [17]. An execution is recoverable if each transaction commits after the commitment of all other transactions from which it reads. An execution avoids cascading aborts if the transaction read only those values that are written by committed transactions or by the transaction itself. An execution is strict if the transaction reads or overwrites a data item after the transaction that previously wrote into it and terminates either by aborting or by committing [18].

In the study of concurrency control, a model of the internal structure of a DBS is shown in Figure 2.3. A transaction manager performs any required pre-processing on database or transaction operations as demanded by the transactions. It is a program or collection of programs that controls the concurrent execution of transactions and makes use of this control by restricting the order in which data manager(operates directly on the database) executes the Reads, Writes, Commits and Aborts of different transactions. The goal of scheduler is to organize these operations so that the resulting execution is serializable and recoverable.

After receiving the operation, the scheduler can take one of three actions [19]:

- **Execute** — It can pass the operation to DM and wait for a result. When DM finishes the executing the operation, it informs the scheduler.
- **Reject** — It can refuse to process the operation, in which case it tells the transaction that its operation has been rejected. This causes the transaction to abort.
- **Delay** — It can delay the operation by placing it in a queue internal to the scheduler.

There are many ways in which the schedulers can be classified [16]. One obvious classification criterion is the mode of database distribution. Some schedulers that have been proposed require a fully replicated database, while others can operate on partially replicated or partitioned databases. The schedulers can also be classified according to network topology. However, the most common classification criterion is the synchronization primitive, i.e. those methods which are based on mutually exclusive access to shared data and those that attempt to order the execution of the transactions according to a set of rules [20, 21]. There are two possible views: the pessimistic view (as explained in the coming section 2.2) in which many transactions will conflict with each other

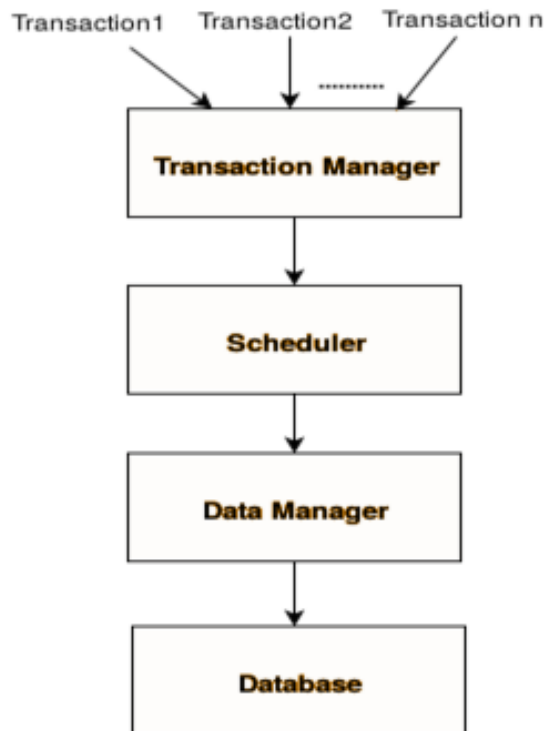


Figure 2.3: Centralized Database System

or the optimistic view (as discussed in the coming section 2.2) that not too many transactions will conflict with each other [22, 20, 23, 24]. Pessimistic methods synchronize the concurrent execution of transactions early in their execution and optimistic methods delay the synchronization of transactions until their terminations.⁵

2.2.1 Pessimistic Concurrency Control

Pessimistic concurrency control is also known as “*locking*”. Locks allow multiple users to safely share a database as long as all users are updating different data at the same time. When locks are used, the locks are placed as soon as any piece of the row is updated and released afterwards. Thus, it is impossible for two users to update a row at the same time. As soon as one user gets a lock, no one else can process that row. This is a safe, conceptually simple approach. The disadvantage is that it requires overhead for every operation, whether or not two or more users are actually trying to access the same record. This overhead is small but adds up because every row that is updated requires a lock. Furthermore, every time that a user tries to access a row, the system must also check whether the requested row(s) are already been locked by another user (or connection). Pessimistic concurrency control is called “*pessimistic*” because the system assumes the worst – it assumes that two users will want to update the same record at the same time, and then prevents

⁵Some additional concurrency control mechanisms are also discussed briefly in this chapter but for this thesis, Optimistic and Pessimistic approaches will be considered and analyzed.

that possibility by locking the record, no matter how unlikely conflicts actually are [15, 25]. An example of pessimistic concurrency control is presented in Table 2.2.

Table 2.2: Example of Pessimistic Concurrency Control

<i>On T1, Entered the statements given below</i>	<i>On T2, Entered the statements given below</i>
<pre>create table t2(a int) engine=soliddb comment='MODE=PESSIMISTIC'; insert into t2 values (1),(2); commit; set autocommit = 0; update t2 set a = 5 where a = 1;</pre>	<pre>set autocommit = 0; update t2 set a = 7 where a = 1;</pre> <p>Note: This query waits for T1 to release locks on table t2</p>

Following are the different types of locking [25, 26]:

- **Simple Object Locking** — It locks each object before writing and unlocks when operation finished.
- **Two Phase Locking (2PL)** — In a transaction, all lock requests precede the unlock requests.
- **Strict Two Phase Locking** — All locks are released when the transaction completes.

2.2.2 Optimistic Concurrency Control

An alternative approach to locking is called “*optimistic*” concurrency control. Optimistic concurrency control assumes that although conflicts are possible, they will be very rare. Instead of locking each record every time that it is used, the software merely looks for indications that two users actually did try to update the same record at the same time [15, 25]. If that evidence is found, then one user’s updates are discarded (and of course the user is informed).

When using an optimistic concurrency control, each time that the server reads a record to try to update it, the server makes a copy of the “version number” of the record and stores that copy for later reference. When it’s time to write the updated data back to the disk drive, the server compares the original version number with the version number that the disk drive currently contains. If the version numbers are the same then no one else has changed the record and the updated value can be written. However, if the new value and the current value on the disk are not the same then it means that someone has changed the data since it was read, and whatever operation was performed is probably out-of-date, so it discards the new version of the data and returns the user an error message. Naturally, each time that updating a record also updates the version number. An example of optimistic concurrency control operation as shown in Table 2.3.

Types of Optimistic Concurrency Control are mentioned below [25]:

- **Backward Oriented Concurrency Control (BOCC)** — This concurrency control mechanism corresponds to the validation of preceding commit while the full-scale execution of the current one.

- **Forward Oriented Concurrency Control (FOCC)** — This mechanism checks whether its write set conflicts with the read set of transactions in their read phase “the *writesets* are only propagated if they do not conflict with current *readsets* of all other active transactions”.

Table 2.3: Example of Optimistic Concurrency Control

<i>On T1, Entered the statements given below</i>	<i>On T2, entered the statements given below</i>
<pre>create table t2(a int) engine=solidb comment='MODE=OPTIMISTIC'; insert into t2 values (1),(2); commit; set autocommit = 0; update t2 set a = 5 where a = 1;</pre>	<pre>set autocommit = 0; update t2 set a = 7 where a = 1;</pre> <p>ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction</p>

2.2.3 No check Concurrency Control

In this alternative approach, no concurrency control checking is done at all. However, normal consistency checking is done, i.e. primary key, foreign key and check constraints are checked⁶ [7]. In this mode, table needs to be designed carefully along with its use afterwards because the concurrent changes could cause unwanted results. When the application uses an update or an insert semantics the data in this table does not need always remain consistent this table. Still, this mode of operation provides a best possible concurrency between transactions and increases the performance because no concurrency checks are performed in almost all phases of the transaction execution. An example of creating no check concurrency is given below:

```
1
2 CREATE TABLE t3(a int) engine=solidb comment='MODE=NOCHECK';
```

2.2.4 Hybrid Concurrency Control

Another approach that is quite semi-optimistic in nature, blocks the operations in some situations where they might violate rules. In other cases, it does not block and the delaying rule keeps on checking till the end of transaction [27, 17, 28]. An example of Hybrid Concurrency Control is given below [16]:

Let's assume that there are three tables A, B, and C. Furthermore, suppose the specification of each of those which is the table A uses pessimistic concurrency control, B uses optimistic concurrency control and C has *nocheck* concurrency control. Now, transaction:

⁶Note: Before the implementation of this thesis, no check concurrency control was implemented in IPBrick database

```

1
2 BEGIN;
3 UPDATE A SET A.B = A.B + 1 WHERE A.ID BETWEEN 21 AND 56;
4 UPDATE B SET B.A = B.A - 1 WHERE B.ID BETWEEN 33 AND 99;
5 UPDATE C SET C.C = C.A WHERE C.ID BETWEEN 77 AND 212;
6 COMMIT;

```

Now this transaction will use pessimistic locking when accessing rows in Table A, optimistic method when accessing rows in the table B and no concurrency control while accessing rows in the table C. Thus, the transaction might wait for locks to be granted during the access of rows in the table A and the transaction might be rolled back at commit time if in the validation phase of rows accessed in Table B are not serialized.

2.2.5 Comparison between Optimistic and Pessimistic Approach

When optimistic concurrency control is used, the user does not find out that there's a conflict until just before user writes the updated data. In pessimistic locking, the user finds out there's a conflict as soon as he/she tries to read the data [22, 12]. To use the analogy with banks, pessimistic locking is like having a guard at the bank door who checks customer's account number when the new user tries to enter; if someone else (a spouse, or a merchant to whom user wrote a cheque) is already in the bank accessing user's account, then the new user can not enter until that other person finishes his/her transaction and leaves. On other hand, optimistic concurrency control allows the user to walk into the bank at any time and try to do business but at the risk that as user is walking out the door the bank, the guard will tell that user's transaction conflicted with someone else's transaction. As a result, it may require to do the transaction again.

Optimistic and Pessimistic concurrency controls differ in another important way besides the time at which conflicts are detected and error messages are issued. Pessimistic locking allows one user to not only block another user from updating the same record but even from reading that record. If the user uses pessimistic locking and gets an exclusive lock then no other user can even read that record. With optimistic locking, however, it is not required to check for conflicts except at the time when it is important to write updated data to disk. If user1 updates a record and user2 only wants to read it, then user2 simply reads whatever data is on the disk and then proceeds, without checking whether the data is locked. User2 might see slightly out-of-date information if user1 has read the data and updated it but has not yet "committed" the transaction [22].

Pessimistic locking gives the user an option that optimistic locking does not offer. It is said earlier that pessimistic locks fail *immediately* – that is, if the user tries to get an exclusive lock on a record and another user already has a lock (shared or exclusive) on that record then it will be informed to the new user that he/she can't get a lock [29]. User might get notified to wait for 30 seconds; this means that if the user initially tries to get the lock and cannot, the server will continue trying to get the lock until either it gets the lock or until the 30 seconds has elapsed. In

many cases, especially when transactions tend to be very short, the user may find that setting a brief wait allows him/her to continue activities that otherwise would have been blocked by locks.

This wait mechanism is applied only to pessimistic locking, not to optimistic concurrency control. There is no such thing as "*waiting for an optimistic lock*" [30]. If someone else changed the data since the time that other user read it, no amount of waiting will prevent a conflict that has already occurred. In fact, since optimistic concurrency methods do not place locks, there is literally no "*optimistic lock*" to wait on. Neither pessimistic nor optimistic concurrency control is "right" or "wrong". When properly implemented, both approaches ensure that user's data is properly updated. In most scenarios, optimistic concurrency control is more efficient and offers higher performance, but in some scenarios pessimistic locking is more appropriate. In situations where there are a lot of updates and relatively high chances of users trying to update data at the same time, the user probably wants to use pessimistic locking. If the odds of conflict are very low (many records and relatively few users, or very few updates and mostly "read" operations) then optimistic concurrency control is usually the best choice. The decision will also be affected by how many records each user updates at a time [9].

2.3 Conclusion

The features of Static and Dynamic Web Pages were deeply analyzed in the chapter and it was concluded that the performance of both of the mechanisms depend on the applications in which they are used. So it is hard to single out one mechanism better than the other. For the web environment with the basics of coding, PHP is probably the right choice to use with the help of a good PHP editor to avoid the Exact Phrasing Problem⁷ (a feature in Perl) and also it has a simpler syntax. With the strong foundations of coding language opens up many options of using complex languages with prototyping. In that case Python is recommended. For this thesis, the author had not this luxury of choosing any of the Server-Side Scripting Language because it was a requirement to use PHP. The appropriate choice of concurrency control procedure depends on the scalability requirements of a given database and how easy it is to write retry loops to re-run failed transactions, etc. So, there is no single right way of achieving concurrency control. However, a definitely wrong way is to ignore concurrency issues and expects that the database takes care of it for the users.

⁷An exact phrase/string is search in another long phrase/string.

Chapter 3

Refinement of IPBrick Solution

For each objective of this thesis, a given solution was adopted from a set of possible ones. This chapter explains why each one was selected, from the ones discussed in the last chapter. The chapter also concludes the presentation of conceptualization facets of the thesis. This makes easier the understanding of the implementation, which is explained in the next chapter.

3.1 Introduction

Literature review in any research work provides a platform to initiate the conceptualization process for a solution. But it stays incomplete if it does not consider the constraints which will be involved in the implementation of it. As this thesis was developed by using the facilities provided at IPBrick, it was important to recognize the factors which can affect the results in a given scenario. All of those factors are considered while modelling the solution for each of the goal of this thesis. The impact of such constraints on the results was predicted in advance so that more outcome should not be expected after the completion of the development phase.

3.2 Administrative Profiles Unit

This section displays the flow charts of the processes which enable an admin of IPBrick private cloud to apply operations on a profile such as: Insert, Delete, View, Modify, Allocate, and Del-Allocate. These flow diagrams led to the development of the mock-ups for the Web Pages pertaining to the Profiles Management Module (See some of the mock-ups in appendix C) and afterward, to their actual ultimate implementations.

Each Profile has two components: Profile Definition and Profile Permissions. The definition facet stores the profile's name and description while the other one manages the information related to a profile's access to various web pages and menus/sub-menus chosen by its creator.

3.2.1 Process of Profile Insertion

Keeping the basic units of a Profile in view mentioned above, the operation of profile insertion will be similar to the one shown in Figure 3.1. At the beginning of Profile Insertion process,

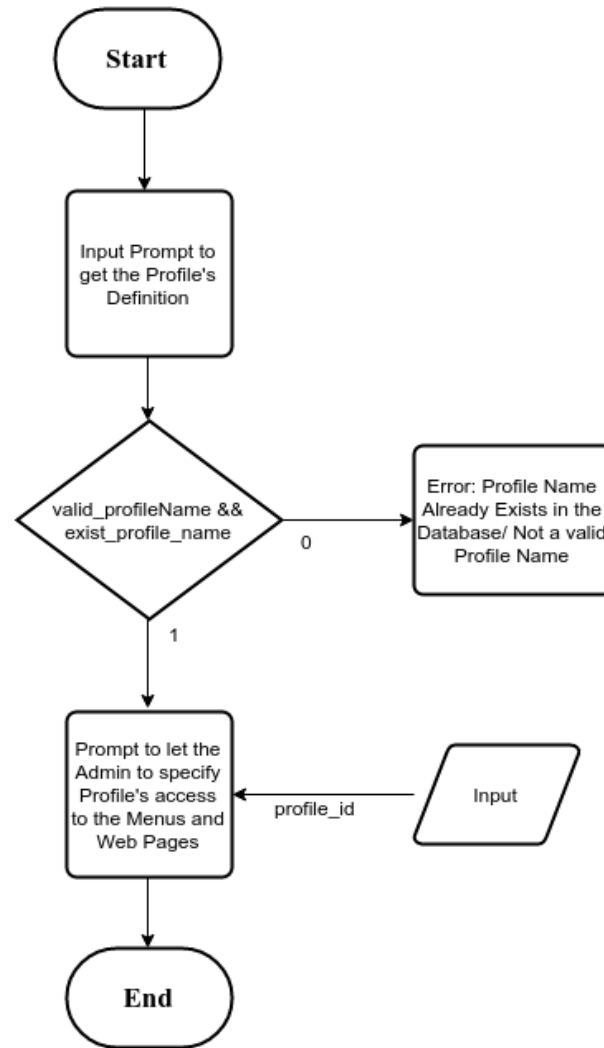


Figure 3.1: Flow chart of the Profile Insertion Process

Profile's name and its description are obtained from the Admin through a user prompt. After these two conditions are applied strictly on the Profile's name must be a valid name(not carrying any non-printable characters) and does not exist in the database. If so, it is stored in database along with the description of the profile. In the proceeding step, Admin is required to specify the list of Menus/Sub-Menus that should be made accessible to this profile by choosing the options given in another prompt. If the Admin does not specify this particular piece of information, then it will not affect the insertion process of the profile. In such case, the user of this Profile will not be able to see any web page or menu of web interface after successfully logging in.

3.2.2 Process of Profile Modification

This process is quite similar to Profile Insertion as it can be seen in the flow diagram as shown in the figure 3.2. In this case, the system has an existing profile data that the admin wants to modify. In the first step, a prompt is shown with the existing definition of the Profile, it is up to the Admin modify it or not. After that the profile's name is validated according to the conditions mentioned in the previous section. This procedure is performed regardless of the fact that it has been modified or not and the process proceeds or terminates beyond this point depends on the validity of the profile's name. If the name is found valid then all of the previous allocation of web pages and menus to this profile will be deleted in the next step. Like the profile insertion process, It is the job of the Admin to specify the new list of web pages and menus/sub-menus should be made accessible to the profile under the operation of modification. If not then the admin of this profile will see a blank Web Page with some alert after logging into the system.

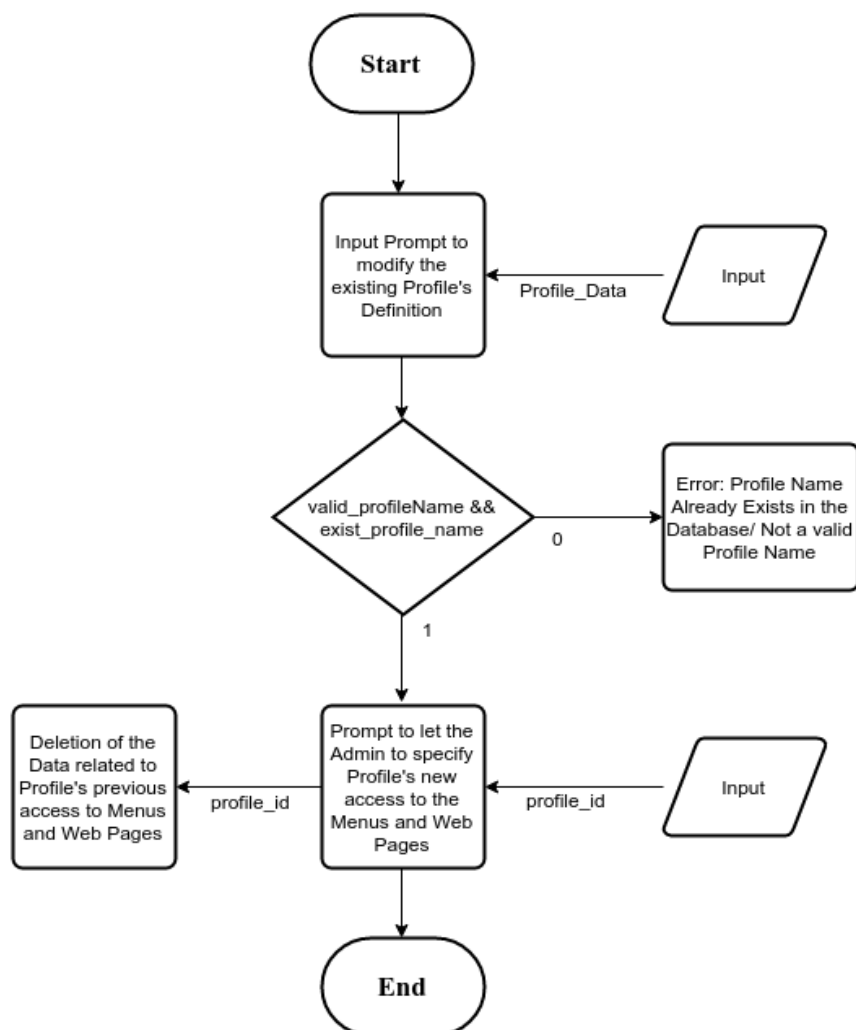


Figure 3.2: Flow chart of the Profile Modification Process

3.2.3 Process of Profile Deletion

In comparison to the previous two processes, this one is quite simple in operation as show in the figure 3.3

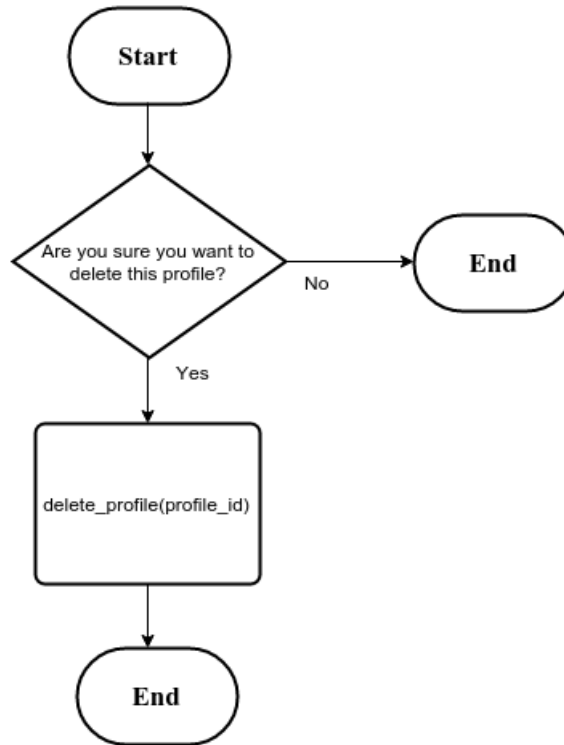


Figure 3.3: Flow chart of the Profile Deletion Process

When this process initiates, it seeks for the administrator's confirmation that he/she is willing to delete this Profile. If the response is no then process terminates without affecting the state of profile at all. Otherwise, all of the profile related information including its definition and permissions will be deleted permanently from the database and system.

3.2.4 Process of Profile Allocation

This process begins with a prompt to get a profile of administrator's choice which he/she wants to assign to some other system user as shown in the initial stage of the flow chart in figure 3.4. In the next step, the admin needs to respond to another prompt through which a user will be selected from the list of system users and subsequently, a profile is assigned to the selected user. If the selected user already contains a profile then the admin needs to confirm whether this user should have a new profile or not. Depending on the its response by the Admin, a corresponding action will be performed as shown in Figure 3.4.

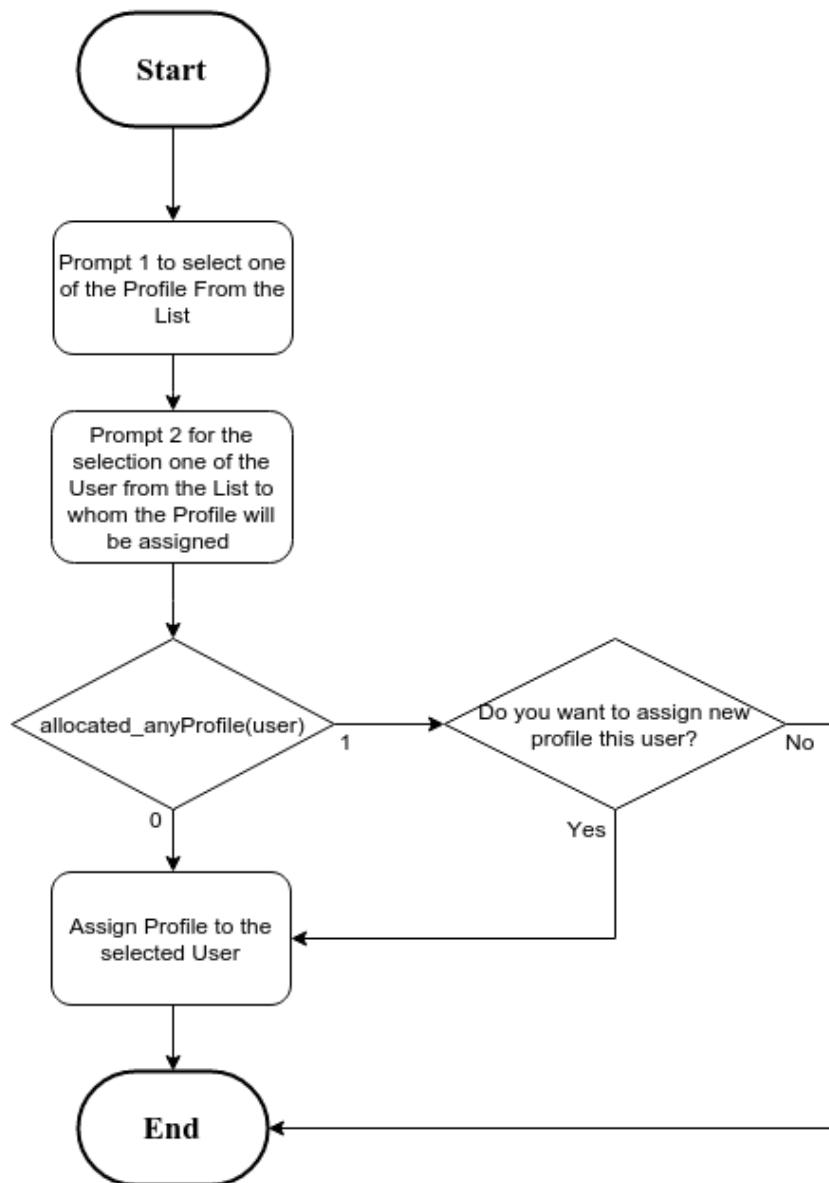


Figure 3.4: Flow chart of the Profile Allocation Process

3.2.5 Process of Profile De-Allocation

This process is a lot more simple than the one for profile assignment, it just cancels the access of a given user to a particular profile. After the completion of this process, profile's list of the allocated users gets updated, along with the one for system users' without any assigned profile.

3.2.6 Process of Profile View

This process is just for displaying the data of any profile stored in the database which can comprise of the basic profile's definition and the list of the users allocated to it. It should be noted that users shown in a view of a given profile have an access to the web interface.

3.3 Multi-Sessions Management

IPBrick cloud-based solution will become operable in the multiple simultaneous sessions very soon. It can give rise to some problems and one of them is the concurrency control in the database of IPBrick. Because operations like update and delete can result into the data conflict and can pose a serious threat to the integrity and the consistency of data in the database. This section discusses the feasibility of various conventional concurrency control schemes in IPBrick database. It also presents the critical analysis which was performed to decide whether any orthodox approach can provide a possible solution for multi-sessions management in a given scenario or not. In the second section, an unconventional approach is also discussed. It became the adopted solution.

3.3.1 Implications in Concurrency Control at Low Level

Concurrency Control is one of the main issues in the database systems as it can result into some severe adverse effects. Due to the strict consistency requirement defined by serializability [15], it can not be considered a viable option in the given system. That's why most of the concurrency control schemes considered in the literature are based on either Optimistic or Pessimistic Concurrency Control. All of those mechanisms have their own pros and cons while considering the formation of IPBrick Database.

2PL with a pessimistic approach has some inherent problems such as the possibility of deadlocks as well as long and unpredictable blocking times. These problems appear too serious in transaction processing, in addition to consistency requirements [28].

Optimistic Concurrency Control methods [18, 30] are especially attractive for the consistency of the data in database systems because they are non-blocking and deadlock-free. Therefore, in recent years, numerous optimistic concurrency control methods have been proposed for databases [16, 18, 30]. Although optimistic approaches such as Time Stamping and Counter Tracking etc have shown to be better than locking methods for database systems in spite of their problem of unnecessary restarts ¹ and heavy restart overhead ². However, because the conflict resolution between the transactions is delayed until a transaction is near its completion, there will be more information available in making the conflict resolution.

The model of IPBrick database is not very suitable for the implementation of any orthodox concurrency control scheme. Other database systems provide storage for the fully committed data in the front and back ends of their databases, user first modifies the data at the front end and then the updates are passed to the back/server end if it satisfies all of the constraints. In this manner, the integrity and consistency of data is protected in those systems. IPBrick database system does not possess user-end and server-end due to which the user always modifies the original copy of the data stored in hard disks. To develop the user-end and server-end IPBrick database now will require lots of time and other resources because it is quite massive in size. Considering all of

¹It occurs when a transaction fails its validation phase is restarted even when history is serializable. Transaction is restarted only if has enough time remaining for meeting its deadline. Otherwise, all transactions are rolled back.

²It is due to the late conflict detection that increases the restart overhead since some near-to-complete transactions have to be restarted because of failed validation

the limitations of IPBrick Database, Table 3.1 shows some new pros and cons of Optimistic and Pessimistic concurrency control along with their inherited ones.

Table 3.1: Pros and Cons of Optimistic and Pessimistic Concurrency Controls

<i>Mechanism</i>	<i>Pros</i>	<i>Cons</i>
<i>Optimistic Concurrency Control</i>	<ul style="list-style-type: none"> • Confirms concurrency control • Does not affect the Read operation 	<ul style="list-style-type: none"> • Modification of All Tables
<i>Pessimistic Concurrency Control</i>	<ul style="list-style-type: none"> • Confirms concurrency control 	<ul style="list-style-type: none"> • Modification of All Tables • Blocks Read Operation too • Affects the relationship between entities in Schema

3.3.2 Solution in the form of Concurrency Control at High Level

Conventional concurrency control mechanisms can not provide a simple solution for Multi-Sessions Management required in the IPBrick system. As a result, an unconventional approach was devised for the accomplishment of this goal. This Multi-Sessions Management scheme works for the update and the delete operations only, which are applied on the data stored in IPBrick database during many concurrent transactions. The process shown in Figure 3.5 begins with the update/delete transaction on the existing data related to an entity stored in the database. This particular transaction makes its entry in another table (Newly created for storing records of the transactions) if there is no other transaction(s) recorded for the same entity or the user with the same session ID as the current one who is willing to make a transaction on a given entity. Otherwise, the user will receive a warning for informing him/her that the desired entity's data is not available for any of the operations except read and it will stay unavailable as long as the other user does not release the hold of it. If the condition is satisfied, the changes will be applied to the database and also the configuration logs of IPBrick updated, for future reference.

This approach is quite simplistic by nature and does not require too complex implementation. It is quite similar to the hybrid technique as it locks the entity's data (already in the middle of some transaction) for other transactions, to ensure its integrity and consistency. Yet, it enables the other users to read the most recently updated entity's data on which some modification operation is also under process simultaneously applied by another user. Not many techniques related to locking provides this option of reading data that is also involved in a write operation except exclusive locking (Complex procedure as it requires to keep a track of all Read and Write Locks along with their management). This newly devised approach can not be compared with the orthodox concurrency control mechanisms in terms of performance because its efficiency has got a limitation and beyond which the performance should not be expected as it requires to store every transaction in the database. But it should not be discarded as it gives instant results without any serious changes in the existing formation of IPBrick database model. Write operation is not considered in this

solution because it is not a threat to the data's consistency and integrity and can be carried without much of the trouble in a multi-sessions based environment.

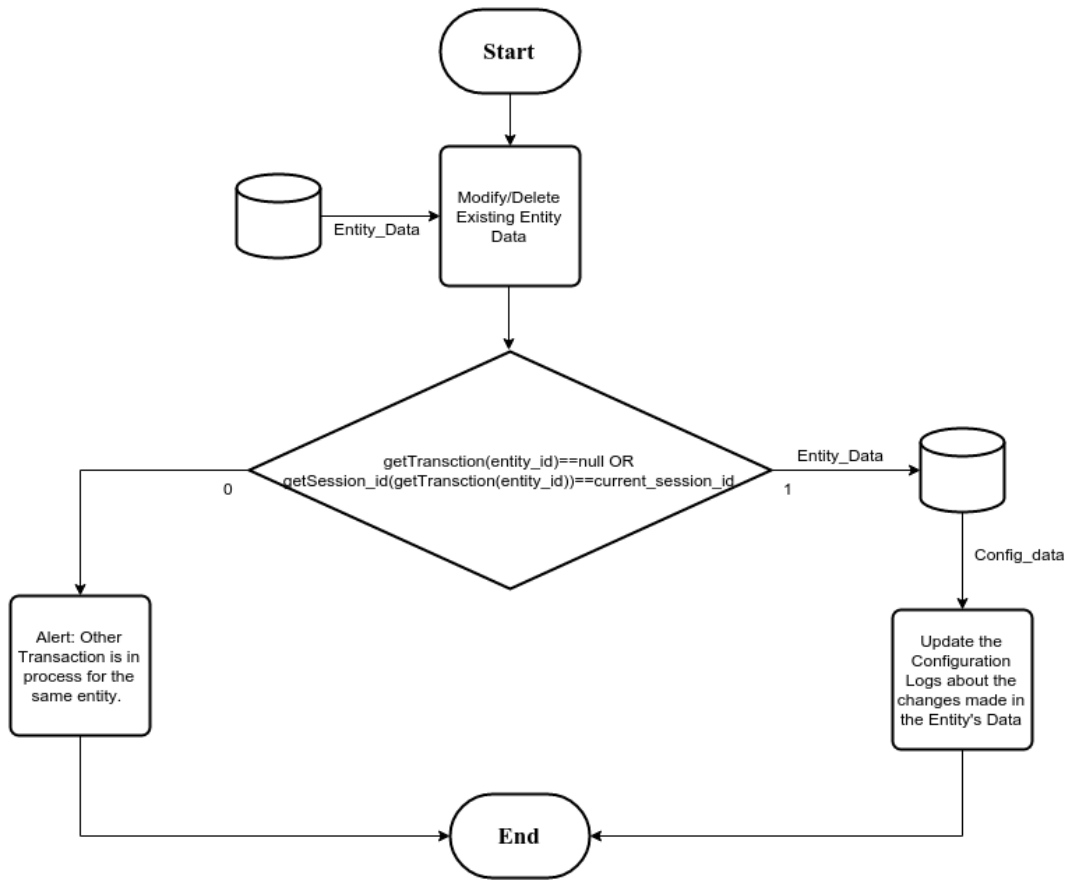


Figure 3.5: Flow chart of the Process handling Multiple Transactions

3.4 Conclusion

In this chapter, the operations related to the Administrative Profiles Unit were discussed in detail with the aid of flowcharts. It had also provided a set of reasons based on which it was concluded that optimistic and pessimistic concurrency control mechanisms are not suitable for multi-sessions management in IPBrick database. An alternative approach was mentioned at the end of this chapter along with the performance detail that can be expected from its efficient implementation.

Chapter 4

Implementation of Modules and Results

In this chapter, author has explained the implementation of the objectives of this thesis which were conceptualized in the previous chapter. It also contains the details of problems faced by the author during the development of each of the goals. Various figures are inserted along with the text to ease the understanding process for the reader. At the end, it concludes with a discussion on the results obtained after the completion of the development phase.

4.1 Introduction

This chapter fully covers the details about the implementations done for the accomplishment of all objectives of the thesis. It explains why and how the solutions were modified during the course of implementation from the ones which were conceptualized in the previous chapter. The web development was mainly performed in PHP language along with some other languages like JavaScript, JQuery and HTML for the addition of interactive features. The software *Netbeans* was chosen for web development part because of its user-friendly interface. Database related implementation was performed in Postgres. Besides that some bash commands were also used for working in the IP-Brick linux-based environment(See the important bash commands in Section D.1 of appendix D).

4.2 IPBrick System

The system of IPBrick operates in a linux-based environment. It is comprised of an overlay of virtual machines associated with various servers which in turn connected to each other via an intranet. Each VM has a web access fully supported by the Postgres-based databases working at the back end of it. The brief details about IPBrick Web Interface and Database are given in the subsequent sections.

4.2.1 Database

The IPBrick DBS contains various databases and the most relevant to this thesis are *systemsoft* and *systemconf*. Some of the important facts related to these two schemas are listed in Table 4.1.

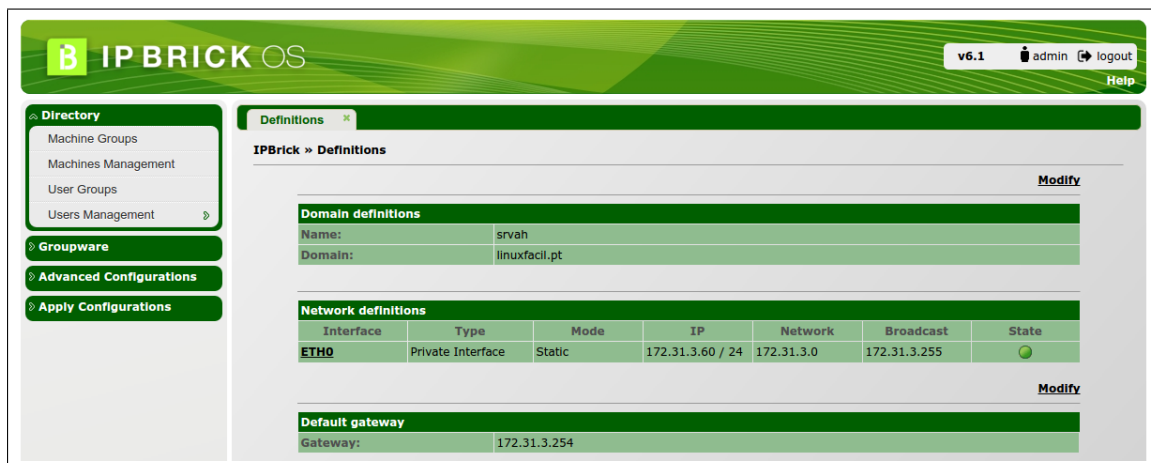


Figure 4.1: Index web page of IPBrick Web Interface

Systemsoft is used to store the data about the web pages and menus related to the IPBrick Web Interface that really helps in navigation between menus to web pages as well from web pages to web pages. *Systemconf* contains the core data of IPBrick relevant to services and different entities, it also stores the information about the defined relationship between the different entities and services such as: allotment of SIP phones to the specific users, the list of the users with email services, etc.

Table 4.1: Facts about the databases *Systemsoft* and *Systemconf*

<i>Names of Databases</i>	<i>Size of Database (MB)</i>	<i>Number of Tables</i>
<i>systemsoft</i>	7.6	161
<i>systemconf</i>	21.5	480

4.2.2 Web Interface

The web interface of IPBrick is comprised of the number of PHP based web pages that allows the admin to manage and utilize the services provided by IPBrick. *class_init* is the one of the most important PHP scripts in IPBrick system, as it can be considered ingress of IPBrick web interface. Important tasks like the inclusion of all PHP classes along with the initialization of their object are performed in the file *class_init*. Other PHP script with a pivotal significance in IPBrick web interface is *corpo* that is helpful in navigating from one web page to other by satisfying all of the conditions specified by the developer(s). *menu* is a PHP script used to display in the IPBrick Web Interface with the menus and sub-menus of all type of services as shown on the left side of Figure 4.1.

4.3 Implementation of Administrative Profiles Unit

The implementation of this objective has been carried out by using the relevant concepts discussed in the chapter 2. This goal had required to make some developments on two fronts of IPBrick: Database and Web Interface. Details for each of them are provided in the following sections.

4.3.1 Inclusion of new Tables

The information related to the following three facets of a profile should be stored in database:

- Profile Definition (Name and Description)
- Profile's access to Menus and their respective Sub-Menus
- Profile's access to Web Pages

As per the categorization of the profile's data, it was considered appropriate to create three tables for each of these parts instead of a single one to store all information in it. It was turned out at the end that in this way, information related to profiles is easily managed.

4.3.1.1 Table to record Profile's Definition

Keeping the operations of Database in view, a profile data should have more fields other than just name and description which can be found in a relational schema mentioned below:

```
profile(idprofile, profile_name, description, profile_type)
```

There is an ID field to serialize the data in the *profile* schema and it also acts as a primary key to avoid the duplication of data stored in Table rows. Attribute *profile_type* is used to easily characterize a profile among types: *Read Only*, *Full Access* and *Custom*. As it was told earlier that IPBrick core data is stored in the database *systemconf*. So Table for profile's definition was made part of *systemconf* as it is shown in figure 4.2.

Some of the important queries applied on this table are shown in Table B.1.1 of appendix B in the form of Relational Algebra.

4.3.1.2 Table to record Profile's Access to Menus

IPBrick provides a variety of the services. For each service there is a menu/sub-menu on the IPBrick web interface and their information is stored in the database *systemsoft*. In order to provide the access of those menus to a profile, a following relational schema made of various fields is needed.

```
profile_menu(idprofile_menu, idprofile → profile, id_menu, menu_level)
```

```

systemconf=# \d profile
          Column          |          Type          |          Modifiers          |
-----+-----+-----+-----+
 idprofile               | integer                | not null default nextval('profile_idprofile_seq'::regclass) |
 profile_name            | character varying(256) |                               |
 description             | character varying(256) |                               |
 profile_type            | integer                |                               |
Indexes:
 "profile_pkey" PRIMARY KEY, btree (idprofile)
Referenced by:
 TABLE "profile_menu" CONSTRAINT "profile_menu_idprofile_fkey" FOREIGN KEY (idprofile) REFERENCES profile(idprofile)
ON DELETE CASCADE
 TABLE "profile_page" CONSTRAINT "profile_page_idprofile_fkey" FOREIGN KEY (idprofile) REFERENCES profile(idprofile)
ON DELETE CASCADE
 TABLE "valida" CONSTRAINT "valida_idprofile_fkey" FOREIGN KEY (idprofile) REFERENCES profile(idprofile) ON DELETE C
ASCADE
systemconf=# SELECT * FROM PROFILE ORDER BY idprofile LIMIT 2;
 idprofile | profile_name | description | profile_type |
-----+-----+-----+-----+
          1 | Default Profile | This profile belongs to the Default SuperAdmin | 2 |
          2 | Profile 0 | Profile for testing only | 2 |
(2 rows)

```

Figure 4.2: *profile* Table along with its fields and entries

Table *profile_menu* shown in Figure 4.3 is based on the relational schema mentioned above. The field *idprofile_menu* is used to serialize the data entries and also to avoid their duplicate copies in this table, similar to the role of *idprofile* in *profile* table. The attribute *id_menu* as its name tells, represents the ID of the particular menu. The last field in Table *profile_menu* is *menu_level* which gives the information about any menu in the menu's hierarchy tree for example, if the sub-menu *User Policies* comes under *User Aborts* which in turns has a parent menu named *Directory*, so the menu levels of the two sub-menus and one menu are denoted as 1.1.1, 1.1 and 1 respectively. As it was discussed earlier that the data related to menus of IPBrick web interface are stored in a table named *menu* of the database *systemsoft* that comprises of the fields *id_menu* and *menu_level*. It is important to note that for the default admin's profile of IPBrick system, this table (stored in *systemconf*) was filled manually and afterwards, it was done automatically for the new profiles. There is another field called *idprofile* acting as a foreign key in Table *profile_menu*, which is used to bring synchronization with the *profile* table. The rows particular to some profile will remain stored in Table as long as the profile will exist in the *profile* table. Otherwise, they will get deleted automatically as *CASCADE* parameter is specified in the definition of *idprofile*.

Few relevant queries applied on this table are shown in Table B.1.2 of appendix B in the form of Relational Algebra.

4.3.1.3 Table to record Profile's Access to Web Pages

Web pages in IPBrick are associated with the menus/sub-menus (*menu_levels* to be precise). So there was a need of the following schema with a right combination of fields that can merge the information related to profile, menu levels and web pages into one body.

```

profile_page(idprofile_page, idprofile → profile, id_page,
menu_level, page_permission)

```

```

systemconf=# \d profile_menu
                                Table "public.profile_menu"
  Column      |          Type          |          Modifiers
-----+-----+-----
 idprofile    |          integer       |
 id_menu      |          integer       |
 menu_level   | character varying(128) |
 idprofile_menu | integer               | not null default nextval('profile_menu_idprofile_menu_seq'::regclass)
Indexes:
 "profile_menu_pkey" PRIMARY KEY, btree (idprofile_menu)
Foreign-key constraints:
 "profile_menu_idprofile_fkey" FOREIGN KEY (idprofile) REFERENCES profile(idprofile) ON DELETE CASCADE

systemconf=# SELECT idprofile_menu, idprofile, id_menu, menu_level FROM profile_menu ORDER BY idprofile_menu LIMIT 5;
 idprofile_menu | idprofile | id_menu | menu_level
-----+-----+-----+-----
              1 |          1 |          1 | 1
              2 |          1 |          2 | 1
              3 |          1 |          3 | 1.1
              4 |          1 |          5 | 1.2
              5 |          1 |          6 | 1.3
(5 rows)

```

Figure 4.3: *profile_menu* Table along with its fields and rows

Figure 4.4 shows a table derived from the schema mentioned above. Similar to the formation of tables *profile* and *profile_menu*, Table *profile_page* has a field *idprofile_page* which serializes the data entries without any duplication. The second field *idprofile* acting as a foreign key synchronizes this table with *profile*. It is a mean of deleting the rows related to a specific profile automatically, if it is removed from the *profile* table. Third attribute *id_page* is for storing the web-pages' IDs defined in the system. The attribute *menu_level* is used to show menu levels respective to the pages specified in the column *id_page*. The field *page_permission* is for the specification of operation which will be applicable to a particular page from a set of permissions¹: insert, delete, modify, view (read all) and all(gives every possible permission on a web page).

```

systemconf=# \d profile_page
                                Table "public.profile_page"
  Column      |          Type          |          Modifiers
-----+-----+-----
 idprofile    |          integer       |
 id_page      |          integer       |
 menu_level   | character varying(128) |
 page_permission | character varying(10) |
 idprofile_page | integer               | not null default nextval('profile_page_idprofile_page_seq'::regclass)
Indexes:
 "profile_page_pkey" PRIMARY KEY, btree (idprofile_page)
Foreign-key constraints:
 "profile_page_idprofile_fkey" FOREIGN KEY (idprofile) REFERENCES profile(idprofile) ON DELETE CASCADE

systemconf=# SELECT idprofile_page, idprofile, id_page, menu_level, page_permission FROM profile_page WHERE idprofile_page BETWEEN 41 AND 46;
 idprofile_page | idprofile | id_page | menu_level | page_permission
-----+-----+-----+-----+-----
              41 |          37 |          412 | 2 | mod
              42 |          37 |          218 | 2.4 | del
              43 |          37 |          219 | 2.4.2 | ins
              44 |          37 |          859 | 2.4.4 | all
              45 |          37 |          266 | 2.4.5 | mod
              46 |          37 |          1013 | 2.4.6 | view
(6 rows)

```

Figure 4.4: *profile_page* Table along with its fields and rows

¹Few of the page permissions are stored in the form of acronyms in *profile_page* table which are insert as 'ins', modify as 'mod', and delete as 'del' while permissions all and view stay as they are

```

systemconf=# \d valida
                                Table "public.valida"
  Column | Type | Modifiers
-----+-----+-----
 login   | text | not null
 pass    | bytea | not null
 type    | integer |
 iduser  | integer | not null default nextval('valida_iduser_seq'::regclass)
 idprofile | integer |
Indexes:
 "valida_pkey" PRIMARY KEY, btree (iduser)
 "valida_login_key" UNIQUE CONSTRAINT, btree (login)
Foreign-key constraints:
 "valida_idprofile_fkey" FOREIGN KEY (idprofile) REFERENCES profile(idprofile) ON DELETE CASCADE

systemconf=# SELECT * FROM valida;
  login | pass | type | iduser | idprofile
-----+-----+-----+-----+-----
 ayaqoob | \x5f0af58bd0814277e6939db9bf5d093f | 2 | 110 | 36
 bhussain | \xba63b5d701415fefda33fd46ec1f46ec | 2 | 111 | 40
 admin | \x91cbfd77ee9162096cf3abb2850807e1 | 1 | 0 | 1
 jhussain | \x639adea504280824086ce4ff2c261bb2 | 2 | 113 | 5
 adhao | \x95falff5da8c98cc7ee6d2e0136dc060 | 2 | 115 | 2
 ahussain | \x401820f4589939915e1e219caa75964b | 2 | 109 | 2
 administrator | \x2d224f5406cbe00c71d30b25de4d0a95017deed2f342449a | 2 | 4 | 46
 srk | \x0a5dfd0e9c2365b9 | 2 | 119 | 37
 jcastro | \xc28cee0ce78e19ebbb639854e0512e54 | 2 | 120 | 37
 sallu | \x4863e09a36e819d106f941bf5040c4df | 2 | 122 | 29
(10 rows)

```

Figure 4.5: Entries in `valida` Table along with a new addition of `idprofile` column

Some of the important queries applied on this table are shown in Table B.1.3 of appendix B in the form of Relational Algebra.

4.3.1.4 Table to record an Administrator's Access to a Profile

Table `valida` as shown in Figure 4.5 existed before the implementation of this thesis. The purpose of this table was to store the credentials of the valid administrators having an access to the web interface of IPBrick. The fields `login` and `pass` are for storing the login-name and password (in the encrypted form) respectively of the users given access to the web interface. The rows in this table are organized on the basis of `login` column as it acts the primary key. Third column `type` maintains the record related the type of access user is having to web interface and at the moment, there are two types of access *remote* and *local* represented as 1 and 2 respectively in the database. Usually, only the default admin falls in the *remote* domain of access to the web interface. Fourth attribute `iduser` stores the users' IDs associated with the corresponding login-name entry of Table. The column `idprofile` representing as a foreign key of `profile` table was a newly added in Table `valida` that enables an administrator to use the IPBrick web interface according to the permissions defined in the profile assigned to him/her.

4.3.2 Inclusion of new Web Pages

In the previous section, we have covered the modifications made in the IPBrick database for the Administrative Profiles unit. This section will discuss the updates made in the web interface of

IPBrick by adding new web pages and modifying the existing ones that are supported by the database at the back-end.

4.3.2.1 Creation of a profile

There are two stages of this process. In the first phase, profile's definition (name and description) are specified while in the second one profile's permissions in terms of its access to menus/sub-menus and web pages are provided.

In Figure 4.6, a HTML element *form* is shown which requires profile's name in *input* tag and profile's description in *textarea* tag² from the administrator. Following actions can take place when the *submit* button is clicked:

- Reloads to the same web page with empty fields if the profile's name is not specified by the admin (the function *reload_all* performs this task and its code is given in Section A.1 of appendix A)
- Stays on the same web pages with the filled fields if the profile's name specified by the admin is not valid (the function *CheckSubmit* performs this task and its code is given in Section A.1 of appendix A)
- Loads to the subsequent action web page after receiving the confirmation of profile's creation from the admin and if the response is negative from the user, action web page is not loaded (the function *CheckSubmit* performs this task and its code is given in Section A.1 of appendix A)

As a result of successful specification of the profile's definition by the admin, an action web page runs without appearing in front of the admin. The script in this web page verifies that the new profile's name does not exist in the *profile* table and also does not contain any unwanted character in it. When the name of the profile is found valid, then the fields *profile_name* and *description* are inserted in Table *profile* along with the auto-increment of B-Tree based *idprofile*. The insert query used for this purpose is "INSERT INTO *profile* (*profile_name*, *description*) VALUES ('NAME', 'TEXT');".

In the second stage, the admin needs to specify the profile's permissions that are its access to menus, sub-menus and web pages. In Figure 4.7, a web page view is shown containing two HTML tables. Table at the top has the static data of profile's definition which was set in the first stage. Table underneath the first one is for the admin to specify the new profile's permissions. As it can be seen in Figure 4.8, there are two columns of Table. The first column contains the instructions for the admin along with some *checkboxes* which make it easy for him/her to select all of the menus and also to choose *view* or *all* permission option for every single web page. In such case, if the user does not want to provide the customized permissions. The selected radio button

²*textarea* is used to get the profile's description, so that it enables the administrator to write a very lengthy text in it

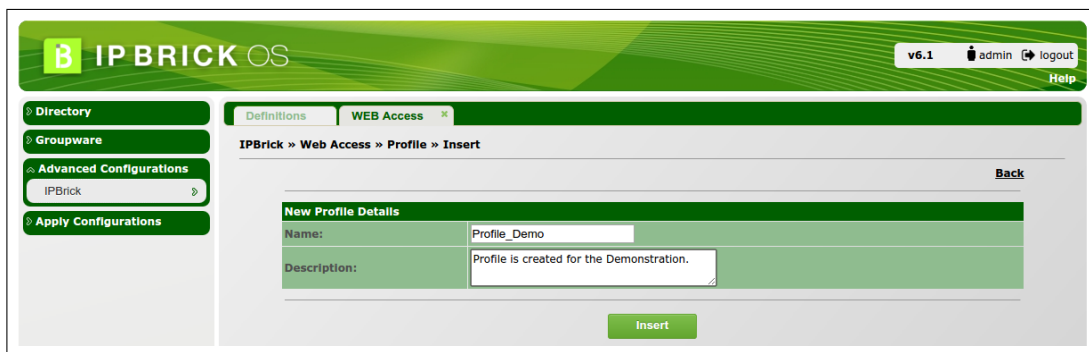


Figure 4.6: Web Page with a prompt for the Profile Definition

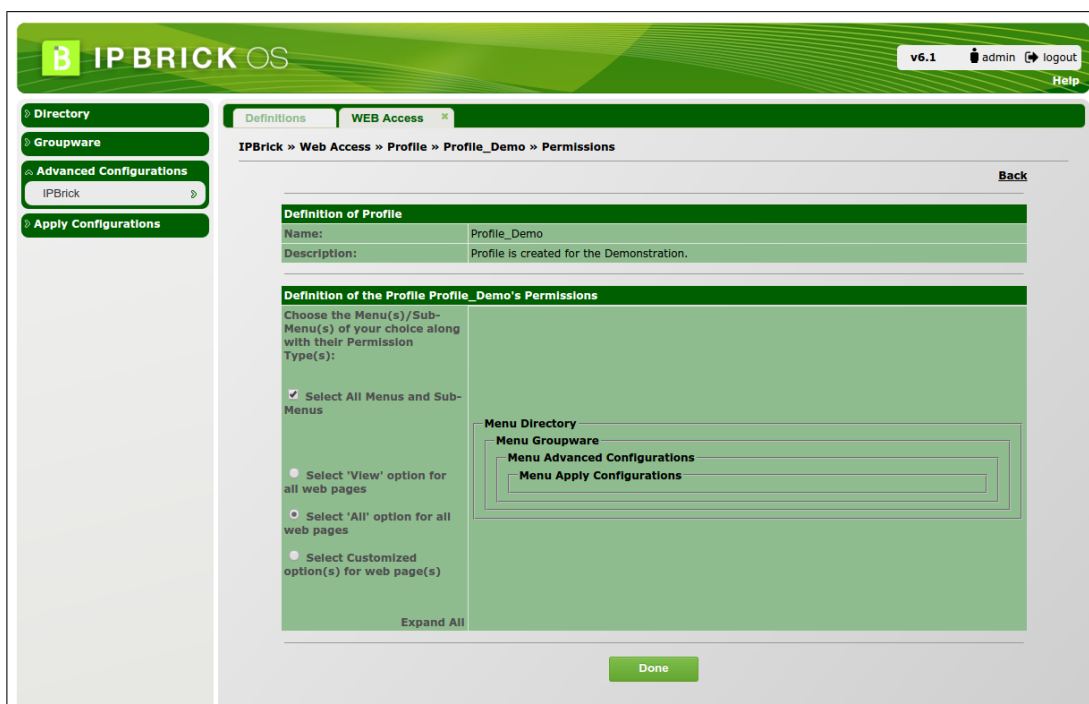


Figure 4.7: Web Page with the Collapsed view of Profile's Permissions

from the given set ultimately ends up as a type of the profile. So, it should be noted that when an admin does not select one of the top two radio buttons given in first column, then the profile type will be set as *custom*, regardless of the fact whether he/she has selected the third radio button in the same column or not (See Section A.2 of appendix A to understand a Javascript based function *Update* implemented for this task). Second column comprises of a tree of menus and sub-menus placing their respective web pages' names underneath them (See Section A.2 of appendix A for understanding the PHP based functions *plotTree* and *printPages* created for this feature).

In the permissions tree, web pages' names are followed by a set of options for operations specified as radio buttons:

- **View** — Acts as a Read-only mode which does not allow to change any aspect of a given web page
- **Insert** — Permits to the use of insert option on a particular web page
- **Modify** — Allows to the use of modify option on a given web page
- **Delete** — Enables to the use of delete option on a particular web page
- **All** — Permits to the use of every possible option like insert, delete, modify and more on a given web page

The admin submits the new profile's permissions after selecting the appropriate options specified in both columns of the second HTML table of Figure 4.8. Then the web system runs another PHP based action script, to make few changes in the relevant tables of the *systemconf* database. Those modifications are mentioned below in the form of queries:

- The update query follows as "*UPDATE profile SET profile_type='TYPE' WHERE idprofile=ID;*" sets the type of an existing profile to 1(*view*) or 2(*all*) or 3(*custom*) depending on what an admin has specified at the time of form submission
- This query "*INSERT INTO profile_menu (idprofile, id_menu, menu_level) VALUES (ID_PROFILE, ID_MENU, 'X.Y.Z');*" inserts the details of the menus and sub-menus, made accessible to a particular as specified by the creator of it
- In-order to insert the data of the selected web pages with respect to their menu levels for a profile is achieved by a query follows as "*INSERT INTO profile_page (idprofile, id_page, menu_level,page_permission) VALUES (ID_PROFILE, ID_PAGE, 'X.Y.Z','PERMISSION');*".

In order to ensure that any selected menu/sub-menu without the selection of its corresponding web page(s) or vice versa can not make an entry in the respective table stored in *systemconf* database, an appropriate condition was applied in the code.

It is recommended to go through the JQuery based code snippet specified in Section A.2 of appendix A which helps to comprehend the interactive features developed for making the permissions selection process more user-friendly.

4.3.2.2 Modification of a profile

This process is very much similar to one for the profile insertion. The only exception is that a profile already exist in the system and it needs to be modified unlike the case in which a profile is created from the scratch. The starting point of profile's modification procedure is shown in Figure 4.9. It is up to the admin to modify the existing definition of the profile or not. If the name of the profile is modified. Then it will be validated just like the way it was done at the time of creation of this profile. As a result of successful validation, its existing entry along

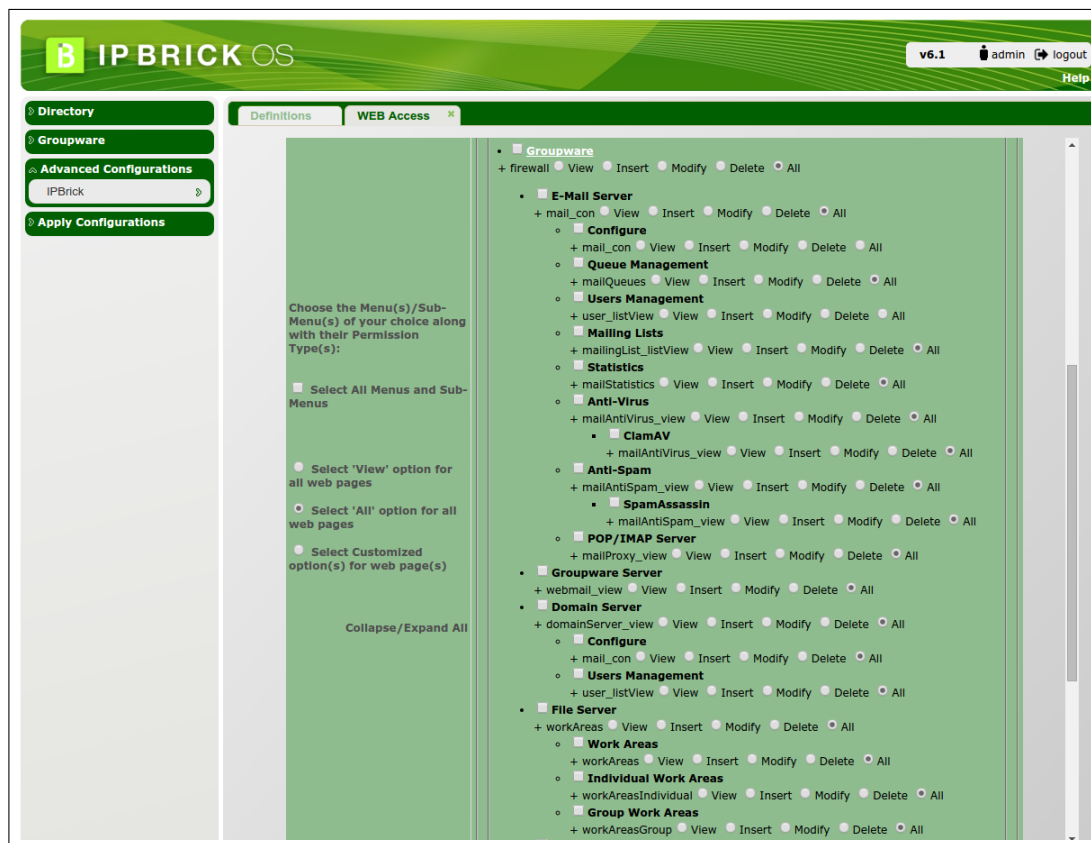


Figure 4.8: Web Page with View of Profile's Permissions

with the description field in the *profile* table gets updated by a query "*UPDATE profile SET profile_name='NAME',description='TEXT' WHERE idprofile=ID;*".

An existing profile owns an access to a set of menus and sub-menus, along with their respective web pages assigned to it previously. So that piece of information needs to be deleted before storing the new one, the following two queries are used to perform this task:

- This query "*DELETE FROM profile_menu WHERE idprofile=ID*" deletes all of the records from Table *profile_menu* for a particular ID of profile
- In-order to delete all of the records from Table *profile_page* for a particular profile ID is achieved by using the query: "*DELETE FROM profile_page WHERE idprofile=ID*"

Rest of the permissions' specification procedure for the profile's modification is to as the one for its insertion. It should also be noted that the same input prompt as shown in Figures 4.7 and 4.8 is used for both profile's insert and update processes.

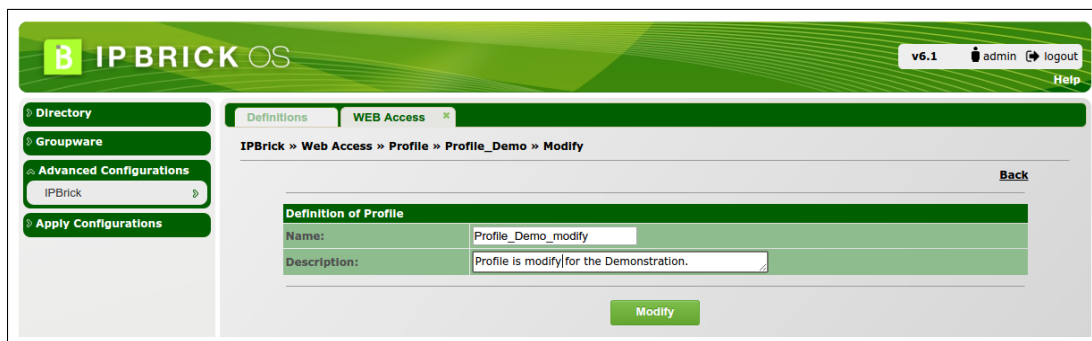


Figure 4.9: Web Page for the modification of an existing Profile

4.3.2.3 Deletion of a profile

Figure 4.10 shows an HTML table containing a profile's data pertaining to its definition. When the *Delete* button is clicked by the Admin, then a javascript based alert appears on the screen. It is for confirmation that the profile's deletion process should execute normally, as the admin does not want to abort. As a result, this query "*DELETE PROFILE WHERE idprofile=ID;*" gets into action and deletes a row from Table *profile* for a particular ID. This change also results into the cascaded deletion of rows specific to the profile ID from Tables *profile_menu* and *profile_menus*.

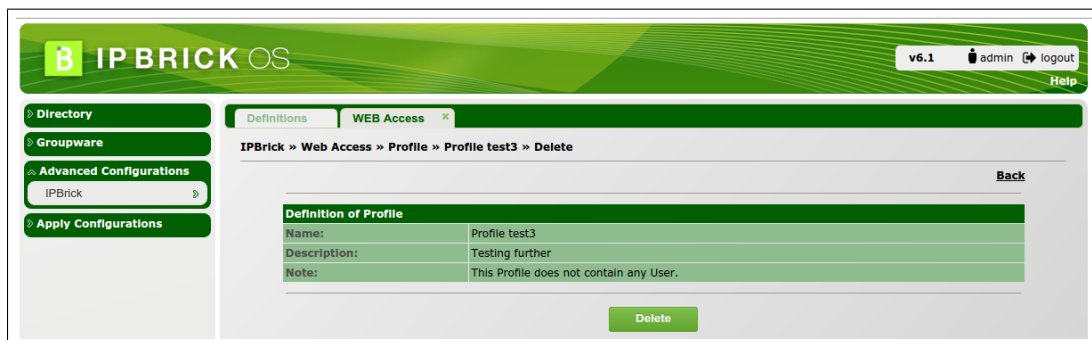


Figure 4.10: Web Page for the deletion of a Profile

4.3.2.4 Allocation and De-allocation of a profile

The first step of the process for allocating a profile to the user is the selection of the profile itself. As shown in Figure 4.11, the profiles' names along with number of the users assigned to each are listed in a drop-down menu. The query "*SELECT COUNT(*) FROM valida WHERE idprofile=ID;*" is used to get the number of users for a particular profile ID from the *valida* table.

When the admin selects one of the profile from the given list and clicks on the submit button, then the web interface crawls to another web page, as can be seen in Figure 4.12.

There are two multi-sized input boxes with two transfer buttons in-between them. The input box on the left hand side contains the list of the users having an access of the selected profile.

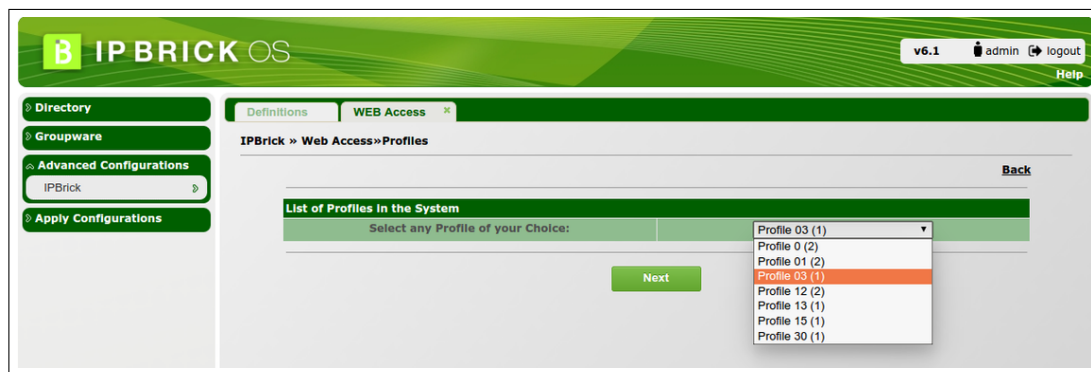


Figure 4.11: Web Page for the selection of a profile from the list

Other one on the right hand side holds the list of user without an access to any profile or having a profile assigned other than the currently selected one. If a user belonging to the list shown at the right side input box already holds a profile then his name is followed by his/her assigned profile name inside the parenthesis. A click on the button with double less-than sign (<<) moves the user's name from the right side input box to the left side input box, as a result of which a user gets access of the chosen profile. The query used for this purpose is `"INSERT INTO valida(login, pass, type, iduser, idprofile) VALUES('LOGIN',\random', TYPE, ID_USER, ID_PROFILE);"`. As user can not hold multiple profiles, so if it already holds one profile then it will be unassigned for the old one and gets allotted a new one. In this particular case, the query `"UPDATE valida SET idprofile=ID_PROFILE WHERE login=LOGIN:"`.

The button with double greater-than sign (>>) works the opposite way in comparison to other button, as it removes the profile's access for some user without assigning any of others. This step can be done by using the query `"DELETE FROM valida WHERE iduser=ID;"`. The Javascript-based functions `AddGrpUser` and `DeleteGrpUser` are working at the back end of these two buttons can be found in Section A.3 of appendix A.

4.3.2.5 De-Allocation of any profile

Previously, it was discussed the removal of the user's access to a specific profile. Now in this section, it will be shown how to do the same for any user with any profile. As it is shown in Figure 4.13, there are two multi-sized boxes with a button in the middle.

When the user with a profile is selected from the right side box, shown in Figure and also the button with a sign ">" is clicked then that user's name is transferred to input box on the right side. Due to this, the user has not longer an access to any profile and can not use the web interface thereafter. The deletion query used for this purpose is `"DELETE FROM valida WHERE iduser=ID;"`.



Figure 4.12: Web Page for the allocation of a user to an existing Profile

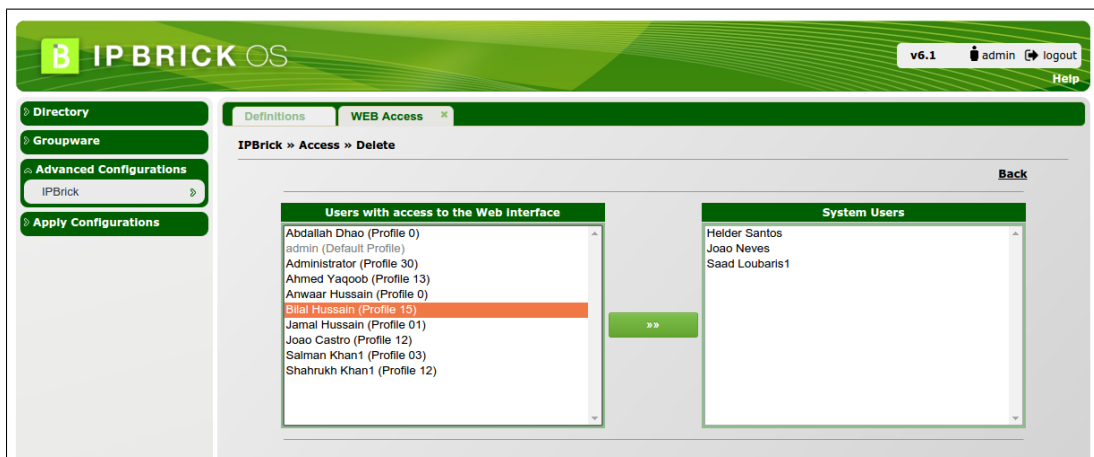


Figure 4.13: Web Page for the removal of a user from accessing an existing Profile

4.3.2.6 View of a profile

Figure 4.14 displays the information pertaining to the definition of a profile in a table element. It also presents the list of users' logins who are provided with its access only if there are any; otherwise, it shows none. The required data for this web page is obtained from the database by using the following queries:

- The query `"SELECT profile_name, description FROM profile WHERE idprofile=ID;"` returns the profile name and description corresponding to profile ID
- For a profile of some specific ID, the list of assigned users' logins is obtained by a query `"SELECT login FROM valida WHERE idprofile=ID"`

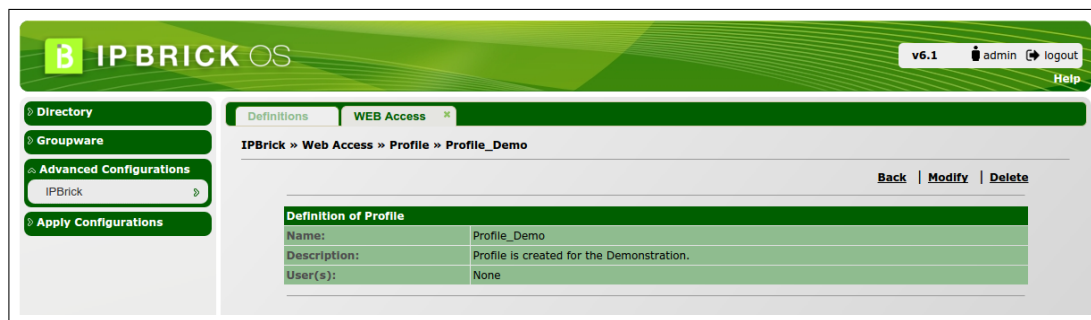


Figure 4.14: Page for viewing an existing Profile

Figure 4.14, there is a set of options: *Back*, *Modify* and *Delete* above *Profile's Definition* table. Profile's definitions and permissions are modified and deleted by navigating in the web pages referenced by the options *Modify* and *Delete* respectively.

4.3.3 Modification of existing Web Pages

Besides the addition of new web pages in IPBrick web interface, some modifications in few of the old PHP scripts were required to be made in-order to successfully integrate the new PHP scripts with the existing system. Details of those changes are given below:

4.3.3.1 *systemDefinition_view*

This web page can be considered as a default web page of Administrative Profiles Unit. This page can be seen in Figure 4.15 which shows three HTML tables: the first for *Access Definition*, second for *User with Access*, the second one for *Profile Management*. There are few other tables which were not considered to be relevant for this thesis.

The following modifications were performed on the HTML tables 2 and 3:

- Previously, the HTML table 2 had two fields: *Login* and *Type of Access* (can be *local* and *remote*). In the context of Administrative Profile Unit, the field *Type of Access* was found irrelevant. Consequently, it was removed and replaced by another column named *Profile*, which shows the profile's name with respect to *Login* of any user. The query used to obtain the name of profile assigned to a user of some particular login is "`SELECT profile_name FROM profile WHERE idprofile=(SELECT idprofile FROM valida WHERE login = LOGIN);`". On top of HTML table 2, there are two text buttons *Modify* and *Delete* which are acting as references to the web pages used for profile modify and delete discussed in Sections 4.3.2.2 and 4.3.2.3.
- Third table shown in Figure 4.15 is used to list the number of profiles by specifying their names and descriptions stored in the database. Such information is obtained by using a query "`SELECT profile_name, description FROM profile;`". Like the second HTML table,

The screenshot shows the IP Brick OS web interface. The header includes the IP Brick OS logo, version v6.1, and user information (admin, logout, Help). The left sidebar contains navigation options: Directory, Groupware, Advanced Configurations (IP Brick), and Apply Configurations. The main content area is titled 'WEB Access' and contains three sections:

Access Definitions

Login:	admin
Password:	*****

User with Access

Login	Profile
adhao	Profile 0
administrator	Profile 30
ahussain	Profile 0
ayaqoob	Profile 13
bhussain	Profile 15
jcastro	Profile 12
jhussain	Profile 01
rsouza	Profile 01
sallu	Profile 03
srk	Profile 12

Profile Management Section

Name	Description
Profile 0	Profile for testing only
Profile 01	This Test is Done to see how long text can be stored in the Description Field by the User (View Only)
Profile 03	Test Textarea
Profile 12	testing the Permissions
Profile 13	Testing Only 13
Profile 15	Testing 15
Profile 30	Without Profile Type

Figure 4.15: Web Page displaying the access details

this one also has one text button *Insert* at its top which navigates to the profile insertion procedure as discussed in Section 4.3.2.1.

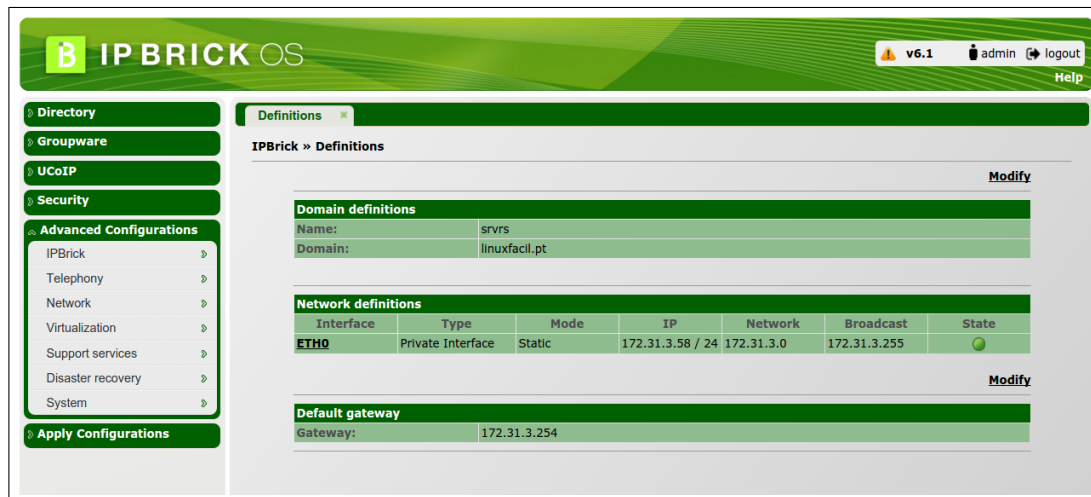
It should be noted that all of the data content which is retrieved from database by using simple queries discussed as above does not get displayed in the HTML tables 2 and 3. Because, all admins are not allowed to view, modify and delete the profiles created by other admins. In order to show the filtered data in Tables 2 and 3 of Figure 4.15, some conditions were applied in the PHP script of *systemDefinition_view* based on the information provided in Table 4.2.

4.3.3.2 corpo

As discussed in Section 4.2.2, all of the web pages in IP Brick interface are accessed through a PHP script named `corpo`. So, applying some conditions in this script was the best possible way of assuring that an Admin can only visit those web pages, which are specified in his/her profile. Beyond that scope, any access to a requested web page is not permitted and receives an alert in response saying that "You are not allowed to access this page!". Important query to achieve this task is "SELECT id_page FROM profile_page WHERE idprofile=ID_PROFILE;". The access

Table 4.2: Privileges of Administrators having Profiles of the different types

<i>Profile Types</i>	<i>Own Profile(s)</i>	<i>Other Profile(s)</i>
<i>Full Access</i>	Can create, view, edit, delete his/her own profile(s)	Can view, edit, delete his/her own profiles created by other Admins except the one belonging to a default admin
<i>Custom</i>	Can create, view, edit, delete his/her own profile(s) if he/she has <i>all</i> permission specified for the web page <i>systemDefinition_view</i> in his/her profile	Can view, edit, delete his/her own profiles created by other Admin having a profile of type Custom or Read Only
<i>Read Only</i>	Can not create his/her own profile	Can only view profiles created by other Admin having the profile types Read Only

Figure 4.16: Possible access to the default web page before the modifications in `corpo` script

of the default web page was denied after the changes had been applied in `corpo` as shown in Figure 4.17 unlike the case presented in Figure 4.16.

4.3.3.3 *menu*

This script displays the menus of services along with the sub-menus for sub-services on IPBrick web interface, as discussed in Section 4.2.2. Previously, it used to retrieve the data of all menus from the database `systemsoft`, which is now changed to obtaining the menus allowed for the profile of some user. A query "`SELECT id_menu FROM profile_menu WHERE idprofile=ID_PROFILE;`" on a table of `systemconf` that returns the menus' IDs particular to a profile ID. The difference in the number of menus can be on the left side of Figures 4.18 and 4.19, which happened due to modifications in the `menu` script.



Figure 4.17: No access to the default web page after the modifications in `corpo` script

4.3.3.4 *default_header*

An idea was established in Section 4.3.2.1 which was that during the creation/modification of a profile, every chosen web page should be specified with a permission from a set of *View*, *Insert*, *Modify*, *Delete*, and *All*. Once that information is stored in the database then the next job is to ensure that an admin should strictly practice only those operations pertaining to his/her defined permission for a web page. This specification is implemented in JQuery by using the function *hide()* and wildcard on selector *href*. After the successful testing of the code snippet given in section A.4 of appendix A, it was placed inside the script named *default_header*. So now this script has two functions: display of a header on top of every web page and show only those operation options on a web page which are related to its specified permission.

4.3.3.5 *class_init*

This script works as a heart of IPBrick web interface because it includes all important classes and also initializes their instances as discussed, in Section 4.2.2. Two PHP classes were also created for this thesis: one for profile and another for transaction (will be discussed in Section of Multi-Sessions Management). The inclusion of those classes along with their instantiations were done in *class_init*, so that their applications can be propagated throughout the system. The lines of code added in the *class_init* can be seen in the snippet given below:

```

1
2 <?php
3
4
5 .....
6
7 include_once ($_path."PHP/IfDBProfile.phpclass"); // IfDBProfile class
8 include_once ($_path."PHP/IfDBTransaction.phpclass"); // IfDBTransaction class
9

```



Figure 4.18: Default web page with All Menus and Sub Menus



Figure 4.19: Default web page with Few Menus and Sub Menus

```

10  . . . .
11
12  $dbprofile = new IfDBProfile ($bd->conn); // Instantiation of IfDBProfile class
    object
13  $dbtransaction = new IfDBTransaction($bd->conn); // Instantiation of
    IfDBTransaction class object
14
15  . . . .
16
17
18  ?>

```

```

systemsoft=# \d pages
                                Table "public.pages"
  Column      |          Type          | Modifiers
-----+-----+-----
 id_page      | integer                | not null default nextval('pages_id_page_seq'::regclass)
 page         | character varying(128) |
 page_old     | character varying(128) |
Indexes:
    "pages_pkey" PRIMARY KEY, btree (id_page)
Referenced by:
    TABLE "liga_pages_links" CONSTRAINT "liga_pages_links_id_page_fkey" FOREIGN KEY (id_page) REFERENCES pages(id_page)
ON DELETE CASCADE

systemsoft=# SELECT * FROM pages WHERE page LIKE '%profile%' ORDER BY id_page;
 id_page |          page          |          page_old
-----+-----+-----
    1376 | profile_ins            | profile_ini
    1377 | profile_ins_act       | profile_ini_extend
    1378 | profile_mod           | profile_modify
    1379 | profile_mod_act       | profile_modify_extend
    1380 | profile_del           | profile_del
    1381 | profile_del_act       | profile_del_extend
    1382 | profile_view          | profile_view
    1383 | profileUser           | profile_existence
    1384 | profilePermissions_ins | profile_permissions
    1385 | profilePermissions_ins_act | profile_permissions_extend
    1386 | profileList           |
(11 rows)

```

Figure 4.20: Table *pages* entries along with the definition of its fields

```

systemsoft=# \d links
                                Table "public.links"
  Column      |          Type          | Modifiers
-----+-----+-----
 id_link      | integer                | not null default nextval('links_id_link_seq'::regclass)
 link_path    | character varying(512) | not null
Indexes:
    "links_pkey" PRIMARY KEY, btree (id_link)
Referenced by:
    TABLE "liga_pages_links" CONSTRAINT "liga_pages_links_id_link_fkey" FOREIGN KEY (id_link) REFERENCES links(id_link)
ON DELETE CASCADE

systemsoft=# SELECT * FROM links WHERE link_path LIKE '%profile%' order by id_link;
 id_link |          link_path
-----+-----
    1384 | ../include/profile/profile_ini.php
    1385 | ../include/profile/profile_ini_extend.php
    1386 | ../include/profile/profile_modify.php
    1387 | ../include/profile/profile_modify_extend.php
    1388 | ../include/profile/profile_del.php
    1389 | ../include/profile/profile_del_extend.php
    1390 | ../include/profile/profile_view.php
    1391 | ../include/profile/profile_existence.php
    1392 | ../include/profile/profile_permissions.php
    1393 | ../include/profile/profile_permissions_extend.php
    1394 | ../include/profile/profileList.php
(11 rows)

```

Figure 4.21: Table *links* entries along with the definition of its fields

4.3.4 Inventory of new Web Pages in database

Every new web page, along with its link in the file system is stored in *systemsoft* database. Without this record, it is not possible to access a web page in IPBrick web interface. For the implementation of Administrative Profile Unit, eleven new web pages were created and their names were registered in Table *pages* as shown in Figure 4.20.

Figure 4.21 shows a table *links* containing the file paths of new web pages. It can be seen in the all paths that each of the new web pages was stored in the common folder named *profile*.

```

systemsoft=# \d liga_pages_links
Table "public.liga_pages_links"
  Column | Type | Modifiers
-----+-----+-----
 id_page | integer |
 id_link | integer |
 link_order | integer | not null
Foreign-key constraints:
 "liga_pages_links_id_link_fkey" FOREIGN KEY (id_link) REFERENCES links(id_link) ON DELETE CASCADE
 "liga_pages_links_id_page_fkey" FOREIGN KEY (id_page) REFERENCES pages(id_page) ON DELETE CASCADE

systemsoft=# SELECT * FROM pages WHERE id_page=1376;
 id_page | page | page_old
-----+-----+-----
 1376 | profile_ins | profile_ini
(1 row)

systemsoft=# SELECT * FROM liga_pages_links WHERE id_link=1384;
 id_page | id_link | link_order
-----+-----+-----
 1376 | 1384 | 2
(1 row)

```

Figure 4.22: Table `liga_pages_links` entries along with the definition of its fields

First, the data related to the names of web pages and their file paths were stored in the respective tables of the database. Then, there was a need for forming the connection between the entries of these two table that was achieved by using the `liga_pages_links`. It can be seen in Figure 4.22, a web page of ID "1376" is linked with a file path "1384".

4.4 Implementation of Multi-Sessions Management

The suggested scheme for Multi-Sessions Management is based on the existing tables: `sessao` and `alteracao` in the database `systemconf`. It is important to note that no change has been made in those tables for the implementation of this solution. For its better understanding, the formation of tables `sessao` and `alteracao` should be learnt.

4.4.1 Table `sessao`

Figure 4.23 shows the definition of `sessao` attributes. The field `id` serializes the `sessao` table and acts as a primary key. As this table deals with sessions, so the session ID is stored in the columns `sessid`. The representation of the user who has initiated a particular session is done by the field `utilizador`. The important statistics about a session are recorded in the columns `inicio` and `expira`.

4.4.2 Table `alteracao`

As a result of some changes made by an admin in the IPBrick web interface, two actions take place. One is the modification in database tables pertaining to the tweaking has been performed at the front end.

The second action is linked with the first one that is updating the configurations logs about the modifications have been done in the database. This step has a preliminary action too and that

```

systemconf=# \d sessao
          Table "public.sessao"
  Column |          Type          | Modifiers
-----+-----+-----
  id     | integer                | not null
  sessid | character varying(32) |
  utilizador | integer                | not null
  inicio | integer                |
  expira | character varying(50) |
Indexes:
  "sessao_pkey" PRIMARY KEY, btree (id)
Referenced by:
  TABLE "transaction" CONSTRAINT "transaction_cc_id_fkey" FOREIGN KEY (id) REFERENCES sessao(id) ON DELETE CASCADE

systemconf=# SELECT * FROM sessao;
 id | sessid | utilizador | inicio | expira
-----+-----+-----+-----+-----
  1 | ddd7d61a259520a1fb9e9ad28e52bcae |      0 |      1 | 1466188138
  2 | 77e963d01f431b739e9bdf9c51543f83 |    109 |      1 | 1466188219
(2 rows)

```

Figure 4.23: Table `sessao` rows along with the definition of its fields

```

systemconf=# \d alteracao
          Table "public.alteracao"
  Column | Type | Modifiers
-----+-----+-----
  servico | text | not null
  alterado | boolean | not null
Indexes:
  "alteracao_pkey" PRIMARY KEY, btree (servico)
Referenced by:
  TABLE "transaction" CONSTRAINT "transaction_cc_servico_fkey" FOREIGN KEY (servico) REFERENCES alteracao(servico) ON DELETE CASCADE

systemconf=# SELECT * FROM alteracao WHERE alterado='t';
 servico | alterado
-----+-----
 ALIASES | t
 MAILFORWARDS | t
 GROUPWARE | t
 APACHE2 | t
 DNS | t
 UTILIZADORGRUPO | t
 GRUPO | t
 PROXY | t
 UTILIZADOR | t
 AQUOTAUSER | t
(10 rows)

```

Figure 4.24: Table `alteracao` rows along with the definition of its fields

is turning the flag(s) to true for the service(s) in `alteracao` table for which the database has been updated. It can be seen in Figure 4.24, `alteracao` table has two columns: `servico` and `alterada`. The former is for keeping the track of services that are modified in a session and the latter is for maintaining the values between 't'(true) and 'f'(false). Services with a true flag in `alterado` column are considered to be changed during the session and their respective logs will get overwritten when the configurations update process will start. Once the detail of those services is stored in the configuration logs, their values in the `alterado` will become 'f' again.

4.4.3 Table *transaction*

As it is discussed in the previous sections that Table *transaction* is going to rely on Tables `sessao` and `alteracao`. In Figure 4.25, the columns `id` and `sessid` are the foreign keys of `sessao` and

```

systemconf=# \d transaction
                                Table "public.transaction"
  Column      |      Type      | Modifiers
-----+-----+-----
 id           | integer        |
 servico     | text           |
 id_entity   | integer        |
 entity      | character varying(20) |
 idtransaction | integer       | not null default nextval('transaction_idtransaction_seq'::regclass)
Indexes:
  "transaction_pkey" PRIMARY KEY, btree (idtransaction)
Foreign-key constraints:
  "transaction_cc_id_fkey" FOREIGN KEY (id) REFERENCES sessao(id) ON DELETE CASCADE
  "transaction_cc_servico_fkey" FOREIGN KEY (servico) REFERENCES alteracao(servico) ON DELETE CASCADE

systemconf=# SELECT idtransaction, id, servico, id_entity, entity FROM transaction;
 idtransaction | id | servico | id_entity | entity
-----+-----+-----+-----+-----
          14 | 1 | AQUOTAUSER |      10003 | user
          15 | 1 | UTILIZADOR |      10003 | user
(2 rows)

```

Figure 4.25: Table *transaction* entries for one session along with the definition of its attributes

alteracao, respectively. The field *idtransaction* in Table *transaction* acts as a primary and also it serialize the data entries without any sort of duplications. Table *transaction* is based on the schema given below:

```

transaction(id_transaction, id → sessao, servico → alteracao,
id_entity, entity

```

In order to approach concurrency control in a more deeply fashion, let's begin with the column *id_entity* which is used to store the actual ID of the entity provided from its respective table. This table also contains an attribute named *entity* which helps to differentiate between the IDs of two entities. Let's consider an example in which there are two transactions: one is related to machine group with *id_entity* equals to 101 and another one belongs to the user group with the same *id_entity*. Now in this scenario, if the column *entity* is not present then it is difficult to differentiate between the two transactions which can lead to problems such as interference between the operations belonging to user and machine categories and errors in the final configuration logs. In most cases, the information stored in the column *entity* will not be required for the normal execution of multiple transactions but still there is a good possibility that it will resolve an awkward situation if it occurs.

4.4.4 Scheme as a Final Solution

This section established a connection between Tables – discussed above – to make easy to understand the working prototype of devised scheme³ used for Multi-Sessions Management. It should be recalled from section 3.3.2 of the Chapter 3 that the concurrency control in this thesis is only applied for the update and the delete operations.

³This solution has been developed for "User Management" service of IPBrick.

```

systemconf=# \d transaction
                                Table "public.transaction"
  Column      |          Type          | Modifiers
-----+-----+-----
 id           | integer                |
 servicio     | text                   |
 id_entity   | integer                |
 entity      | character varying(20) |
 idtransaction | integer                | not null default nextval('transaction_idtransaction_seq'::regclass)
Indexes:
 "transaction_pkey" PRIMARY KEY, btree (idtransaction)
Foreign-key constraints:
 "transaction_cc_id_fkey" FOREIGN KEY (id) REFERENCES sessao(id) ON DELETE CASCADE
 "transaction_cc_servico_fkey" FOREIGN KEY (servico) REFERENCES alteracao(servico) ON DELETE CASCADE

systemconf=# SELECT idtransaction, id, servico, id_entity, entity FROM transaction;
 idtransaction | id | servico | id_entity | entity
-----+-----+-----+-----+-----
          14 | 1 | AQUOTAUSER |      10003 | user
          15 | 1 | UTILIZADOR |      10003 | user
          16 | 2 | AQUOTAUSER |      10010 | user
          17 | 2 | UTILIZADOR |      10010 | user
(4 rows)

```

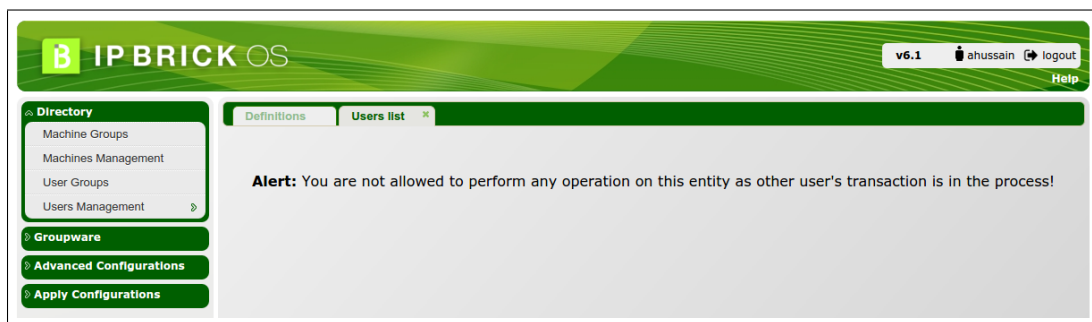
Figure 4.26: Table *transaction* entries for two sessions along with the definition of its attributes

Figure 4.27: An alert for the second Admin if he wants to modify the same data as the first one

Let's proceed with the understanding of Multi-Sessions Management Scheme. Consider a scenario in which two admins are trying to modify two different user's data stored in the database. In this case, the admins' respective transactions will get committed smoothly without any problem as can be seen in Figure 4.26. For this purpose, the query `"INSERT INTO TRANSACTION(id, servico, id_entity, entity) VALUES(ID, 'TEXT', ID_ENTITY, 'TEXT');"` is used for all inventories made in the *transaction* table. One more point relevant to this scenario should be considered that a transaction related to a particular user in *transaction* table can only be recorded once. If another transaction is created for the same user in a session then it will be not be stored in Table *transaction* twice or more although, Table pertaining to that user will always carry its latest entry.

Another case shown in Figure 4.25, in which the data of user with ID equal to "10003" are being updated by an admin. Now, if another admin wants to modify the data of the same user then he/she will not be allowed to do so and will receive a notification shown in Figure 4.27.

To generalize the case in which more than one transactions are trying to modify the data of the same entity and how each of the transaction terminates ultimately – without affecting the data consistency and integrity – can be observed in Table 4.3.

Table 4.3: Concurrency Control between N transactions related to the same entity's data

<i>Transaction 1</i>	<i>Transactions 2 to N+1</i>
$T_1 \rightarrow update(A)$	$T_2 \rightarrow update(A)$
	Aborts
	$T_3 \rightarrow delete(A)$
	Aborts
	...
	$T_N \rightarrow update(A)$
	Aborts
Commits	$T_{N+1} \rightarrow delete(A)$
	Commits
	...

A simple query "*SELECT id, id_entity, entity WHERE servico= 'TEXT';*" is used to get the transaction data for a particular *servico*. A new transaction will always get rejected for the different values of column *id* if its *id_entity* and *entity* are equal to other transaction(s) that had already been stored in Table *transaction*.

When an admin holding a session applies the configurations in his/her system for updating the logs with some new changes then all rows related to that session will be deleted from *transaction* table by using a query "*DELETE FROM transaction WHERE id=ID;*". Another action will also be performed simultaneously which is *alterado* flag will be changed from '*t*' to '*f*' for the *servico* involved in the deleted transactions only if they are not used in any other entry of *transaction* table. Otherwise, no changes will be made in *alteracao* table. Finally, the column *alterado* of Table *alteracao* will become '*f*' for all values of *servico* when *transaction* table will run out of its every single row.

4.5 Extra Work

In the initial phase of this thesis, some extra objectives were also set besides the actual ones. Their implementation was completely dependent on the availability of time at the end. Luckily, some extra development work was also performed due to the early accomplishment of main goals.

4.5.1 Renaming of Web Pages' Names in the Database

Over the years, hundreds of web pages were added in IPBrick system without having a proper nomenclature for their names. Due to this reason, web pages' names are full of inconsistencies such as the existence of names in various languages, the use of meaningless abbreviations in names, etc. That makes difficult for developers and especially for customers to understand the IPBrick sitemap easily.


```

systemsoft=# \d pages
                                Table "public.pages"
  Column      |          Type          | Modifiers
-----+-----+-----
 id_page     | integer                | not null default nextval('pages_id_page_seq'::regclass)
 page        | character varying(128) |
 page_old    | character varying(128) |
Indexes:
    "pages_pkey" PRIMARY KEY, btree (id_page)
Referenced by:
    TABLE "liga_pages_links" CONSTRAINT "liga_pages_links_id_page_fkey" FOREIGN KEY (id_page) REFERENCES pages(id_page)
ON DELETE CASCADE

systemsoft=# SELECT * FROM pages ORDER BY id_page LIMIT 5;
 id_page |      page      |      page_old
-----+-----+-----
      1 | group_listView | grupo_ver_lista
      2 | group_view     | grupo_ver
      3 | group_ins      | grupo_inserir
      4 | group_ins_act  | grupo_inserir_inser
      5 | group_mod      | grupo_alterar
(5 rows)

```

Figure 4.28: Table *pages* containing pages' names based on a new nomenclature in *page* column

In order to bring uniformity in the web pages' names, a new nomenclature was defined in English language as "*FullPageName[_TypeOfPage][_Action]*" based on which more than 850 out of 1385 web pages were renamed. Few of those can be seen in the column *page* of *pages* table show in Figure 4.28. Previously, the web pages' names were so out of order that it was very difficult to build a script which can rename the web pages easily. As a result, it was done manually one by one. Although, this task is not very technical in nature besides the use of some *SELECT* and *UPDATE* queries but it will prove to be fruitful for IPBrick in a long-term.

4.5.2 Run-Time Configurations

The existing IPBrick solution does not immediately put the changes made by an admin into effect until another admin having some sort of priority does not update the configuration logs with his/her changes. The second admin and rest of others accessing the system after him/her are made to wait in a queue because of the complicated way of operation. To address this problem, IPBrick has developed an external module for an older version of its solution to allow the configurations to be applied on a run-time basis. It was set as an additional goal to implement this feature similar to the previous with some improvements for the latest version of IPBrick solution.

Previously implemented web service was based on SOAP that allowed to develop it in an interoperable fashion by defining some rules for sending and receiving RPC such as the-XML based structure of the request and response (See Section D.3 of appendix D). Those RPCs were exchanged between a SOAP Client and a SOAP Server according to specifications defined in the WSDL Document Style Model(Message-Oriented style). WSDL is a XML document that provides meta-data for a SOAP service (See the structure of WSDL in Section D.4 of appendix D). They contain information about the functions or methods the application makes available and what arguments to use [31]. By making WSDL available to the clients of web service, gives them the definitions they need to send valid requests precisely how it is required to be. SOAP can work properly without WSDL but it can offer a lot if used. WSDL tells how to connect with

Table 4.4: Parameters of the function *applyAddUserRuntime*

<i>Parameter</i>	<i>Required Value</i>
<apiAccessLogin>	Valid Login of the admin who is trying to use the web service
<apiAccessPass>	Valid Password of the admin who is trying to use the web service
<name>	Name of the new user
<login>	Login of the new user (will be stored in lowercase only)
<quota>	Email Quota of the new user
<area>	1 for home1, 2 for home2
<server>	0 for local, 2 for remote
<password>	Password for user authentication
<employeenumber>	any number
<departmentnumber>	any number
<roomnumber>	any number
<pager>	any number
<employeeetype>	any number or text
<businesscategory>	any number or text

communication server and SOAP provides the communication messages.

In this thesis, it was attempted to add the user to IPBrick System on run-time basis by using SOAP along with WSDL. For this purpose, the entities like client and server were created and they communicated with each other through Request and Response mechanism. The main function of this service is *applyAddUserRuntime* which adds the user on run-time and does not need the admin's intercession in applying configurations. An aspect of authentication was also introduced in this web service which requires the credentials: *apiAccessLogin* and *apiAccessPass* in the form given below:

```

1
2 <?php
3
4 $apiAccessLogin = "969b14b217e2d91ad075a6";
5 $apiAccessPass = "adaeaa7x3a5400c195e499";
6
7 ?>
```

In the context of IPBrick solution, the function *applyAddUserRuntime* requires the parameters in the order as presented in Table 4.4.

The binding of WSDL with SOAP can be quite cumbersome, as it requires to write the specific tags – quite long in length – by hand.. This task was simplified by using NuSOAP⁴ which created a WSDL file automatically.

The code of *server* script is mentioned below:

⁴NuSOAP is a set of PHP classes – no PHP extensions required – that allow developers to create and consume web services based on SOAP 1.1, WSDL 1.1 and HTTP 1.0/1.1

```

1 <?php
2
3 require_once "nusoap.php";
4
5 function applyAddUserRuntime($apiAccessLogin, $apiAccessPass, $name, ...) {
6     ....
7     ....
8 }
9
10 $server = new soap_server();
11 $server->configureWSDL("applyAddUserRuntime", "urn:applyAddUserRuntime");
12
13 $server->register('applyAddUserRuntime',
14                 array('apiAccessLogin' => 'xsd:string',
15                       'apiAccessPass' => 'xsd:string',
16                       'name' => 'xsd:string',
17                       'login' => 'xsd:string',
18                       'quota' => 'xsd:int',
19                       'area' => 'xsd:int',
20                       'password' => 'xsd:string',
21                       'employeenumber' => 'xsd:string',
22                       'departmentnumber' => 'xsd:string',
23                       'roomnumber' => 'xsd:string',
24                       'pager' => 'xsd:string',
25                       'employeeetype' => 'xsd:string',
26                       'businesscategory' => 'xsd:string'),
27                 array('result' => 'tns:StdResult'),
28                 'uri:'.NUSOAP_NAME_SPACE,
29                 'uri:'.NUSOAP_NAME_SPACE.
30                 '/applyAddUserRuntime','rpc',
31                 'encoded',
32                 'Insert Users on Run-Time');
33
34 $server->service($_HTTP_RAW_POST_DATA);
35
36 ?>

```

As it can be seen in the code, a new instance of the `soap_server` class is instantiated which calls to `configureWSDL()` so that WSDL file can be created. This function has two arguments, former is for the name of the service and later belongs to the namespace of the web service. The object of `soap_server` class calls another function `register` which builds the content of WSDL file. It has following argument [32]:

- First argument of this function is the name of function (`applyAddUserRuntime`) called by the client of this web service
- Second argument is an array which contains all of the input parameter acquired by function

- Third argument is also an array which defines the return value along with its datatype
- *rpc* defines the type of call (this could be either *rpc* or *document*)
- *encoded* defines the value for the use attribute(Either *encoded* or *literal* can be used)
- The last parameter is a documentation string that describes what the *applyAddUserRuntime* function does

A file with the name *server.wsdl* was create automatically when the URL⁵ was pointed in the brower. At the client side, this file is accessed through the constructor of *SOAPClient* class. Inside the client file, a call for the function *applyAddUserRuntime* is made. The return value should be a positive number in the case of its successful operation or -1 if some error occurs. Some portion of the script that was created for the client is given below:

```

1
2 <?php
3
4 $name = $argv[1];
5 $login = $argv[2];
6 $quota = $argv[3];
7 $area = $argv[4];
8 $server = $argv[5];
9 $password = $argv[6];
10 $employeenumber = $argv[7];
11 $departmentnumber = $argv[8];
12 $roomnumber = $argv[9];
13 $pager = $argv[10];
14 $employee type = $argv[11];
15 $businesscategory = $argv[12];
16
17 $uri= "urn:". $class ;
18 $wsdlUrl= 'https://172.31.3.60/runtime/server.php?module=' . $class . '&wsdl' ;
19 $location= 'https://172.31.3.60/runtime/server.php?module=' . $class ;
20
21 $client = new SOAPClient($wsdlUrl ,
22 $client->applyAddUserRuntime($apiAccessLogin , $apiAccessPass , $name , $login ,
    $quota , $area , $server , $password , $employeenumber , $departmentnumber ,
    $roomnumber , $pager , $employee type , $businesscategory) );
23
24 ?>

```

Due to the incomplete implementation of the run-time module, none of the expected value is obtained. But this work can be carried forward and can lead to the resolution of the problem at hand.

⁵<https://172.31.3.60/runtime/server.php?&wsdl>

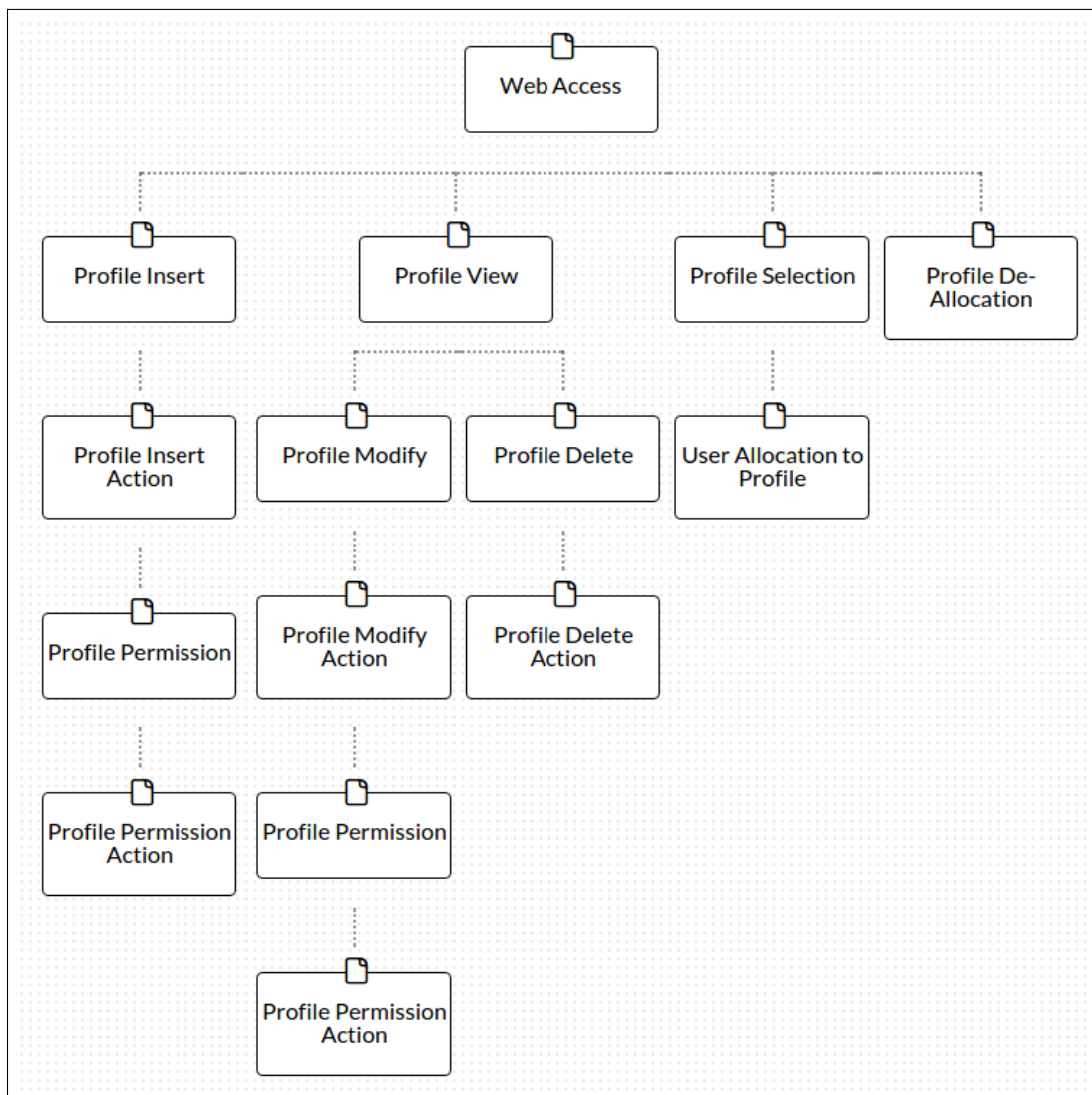


Figure 4.29: Sitemap of Administrative Profiles Unit

4.6 Results

The performance of implementation performed for this thesis can be judged at two ends. On front end, it can be validated whether all of newly developed web pages are well-connected with the existing IPBrick Web Interface or not. While at back end, it can be tested how quickly database responds to the designed queries.

4.6.1 Performance Analysis of Administrative Profiles Unit

Figure 4.29 shows how Administrative Profiles Unit is integrated with the web interface of IP-Brick. Regardless of the operation performed on profile by navigating through a designed set of web pages, the whole system is capable enough to crawl back to the web access page without straying to a dead end [33, 34, 35].

Table 4.5: Data Set of Test Scenario for Administrative Profiles Unit

<i>Given Data</i>	<i>Quantity</i>
Number of Profiles in <i>profile</i> table	5
Number of valid Admins in <i>valida</i> table	7
Number of Menus and Sub-Menus in <i>profile_menu</i> table	56
Number of Web Pages in <i>profile_menu</i> table	58
Number of Menus and Sub-Menus in a Small-Sized Profile	9
Number of Menus and Sub-Menus in a Medium-Sized Profile	17
Number of Menus and Sub-Menus in a Large-Sized Profile	35
Number of Web Pages in a Small-Sized Profile	8
Number of Web Pages in a Medium-Sized Profile	16
Number of Web Pages in a Large-Sized Profile	33

Table 4.6: Execution Time of the different Operations in the Administrative Profiles Unit

<i>Applied Operations on a Profile</i>	<i>Time for a Small-Sized Profile (ms)</i>	<i>Time for a Medium-Sized Profile(ms)</i>	<i>Time for a Large-Sized Profile(ms)</i>
Insert	5.37	9.63	18.15
Modify	6.49	10.51	19.29
View	1.37	2.58	4.33
Delete	1.39	1.44	1.57
Allocation	101.51	101.51	101.51
De-Allocation	.84	.84	.84

A website or any set web pages' seamless property becomes more prominent if its operations related to the database are performed quickly at its back-end. A very simple group of queries was used in order to make all aspects of Administrative Profiles Unit fully functional without consuming much time. But when Tables start getting populated with a large bulk of data then those simple queries take more execution time for the same tables because of search time overhead. As a result, the user of a website or its unit starts to experience the delays between the loading of different web pages. There could be other reasons too for those hold ups such as the use of some powerful interactive elements in web scripts and few more. But, still, the heavy database procedure is the main cause of introducing some jitters in the flow of the website.

Table 4.5, a test case is prepared to see the database response time for each process involved in Administrative Profiles Unit. In order to create a realistic scenario, Tables *profile*, *profile_menu* and *profile_page* were provided with a number of rows just to involve searching process when some query is applied on them. In this test case, three profiles of three different scopes: Small, medium and large were created and each of those profiles composed of different number menus and web pages from others.

The possible operations which can be performed on a profile are listed in the column "Applied Operations on a Profile" of Table 4.6. Each operation involves the number of queries from a set *INSERT*, *SELECT*, *UPDATE* and *DELETE* depending on its functionality. The values in milliseconds were obtained by adding the calculated execution time for all individual queries involved in a

Table 4.7: Data Set of Test Scenario for Multi-Sessions Management Scheme

<i>Given Data</i>	<i>Quantity</i>
Number of rows in <i>sessao</i> table	2
Number of rows in <i>alteracao</i> table	174
Number of transactions in <i>transaction</i> table	5
Number of transactions to be inserted in <i>transaction</i> table	2
Number of transactions to be deleted in <i>transaction</i> table	2

process of some profile. Run-time for any type query can be determined in milliseconds by using the Postgres command "*EXPLAIN ANALYZE query*";".

As it can be seen in Table 4.6, the execution time of processes: Insert, Modify, View and Delete is directly proportional to the size of a profile. Insertion and Modification of a Profile are quite similar operations but latter takes more time than former because it involves the search time overhead. The process "Profile Allocation" consumes more time than any of other processes because it involves the encryption of password assigned to the user. A noticeable fact about Allocation and De-allocation of a profile is that the size of profile does not matter as the execution time remains the same for all.

4.6.2 Performance Analysis of Multi-Sessions Management Scheme

The efficiency of the scheme developed for Multi-Sessions Management is heavily dependent on the response time of queries made on the database. For the purpose of testing, a scenario was created by keeping a pragmatic approach in view that is to provide a set of rows in the given tables of the database as it can be seen in Table 4.7.

In this test case, two transactions will be inserted into Table *transaction* and then the two will be deleted from the same table. The transaction insertion and deletion processes do not involve only one query on *transaction* table for each process, but they also execute few more queries related to *alteracao* along with *sessao* before going for the execution of final one. Runtime in milliseconds for each applied operation on transactions is calculated by adding the execution time of all queries involved in it. The command used to determine the execution time of any query is "*EXPLAIN ANALYZE query*";".

It can be observed in Table 4.6 that the time taken for the successful insertion of transactions in *transaction* is greater than the unsuccessful ones. Because once the transactions satisfy the specified criteria then they can make their entries in Table *transaction* which affects the final sum of time in milliseconds. For the process of transactions deletion, it is checked that the transactions to be deleted from *transaction* table do not contain any *servico* involved in other pending transactions. If there are no remaining transactions in *transaction* table besides the ones involved in the process of deletion then the execution time will be less as there will be no search time involved in it. Otherwise, it will be greater in value. Both cases related to the deletion process were tested and can be found in Table 4.6.

Table 4.8: Execution Time of the different Operations in the Multi-Sessions Management Scheme

<i>Applied Operations on Transactions</i>	<i>Run Time</i>	<i>Remark</i>
<i>Insertion</i>	2.134ms	Successful without resulting into Data Conflict
	1.084ms	Not Successful as a result Data Conflict occurs
<i>Deletion</i>	4.538ms	Successful while having some pending transactions in the queue
	3.48ms	Successful while having no pending transactions in the queue

4.7 Conclusion

In this chapter, the front and back end implementations performed for Administrative Profiles Unit were discussed in detail with the support of relevant diagrams. It had also explained how the multi-sessions management scheme was developed for "User Management" service and highlighted its bright prospect of getting it implemented for all services of IPBrick. At the end of the chapter, some test cases relevant to the implementations were presented in order to discuss their efficiency.

Chapter 5

Conclusion and Future Work

This chapter contains the concluding remarks of the author related to the thesis' topic and the research performed by him during its development. It specifies the extents up to which the goals of dissertation were achieved. At the end, it also discusses how further improvements can be brought in the results obtained.

5.1 Accomplished Goals

“If you follow reason far enough it always leads to conclusion that are contrary to reason.”

Samuel Butler

The main aim of this thesis was to enhance IPBrick solution in a way that it can meet the modern-day requirements of its customers without making any drastic changes in it. For this purpose, few objectives were set initially, conceptualized during the course of thesis and accomplished later. Now, IPBrick Administrators can create new profiles and assign those to other new Admins of IPBrick Private Cloud. These profiles can be viewed by all of the admins with an access to IPBrick web interface but few of them can modify and delete them based on the specifications of their profiles. By virtue of Administrative Profiles Unit, the user's access to profiles can also be revoked anytime by a default administrator(s) of IPBrick system. Besides applying the essential operations on the profiles, some procedures were also developed to ensure that an admin of some particular profile could not use any menus and sub-menus along with their respective web pages beyond the scope of his/her profile's definition. Web pages' permissions were successfully applied on each web page which helped to control the admins' actions such as no access to the web pages not belonging to the domain of their profiles.

The second major objective of this thesis was to develop an efficient scheme for multi-sessions management which can offer some sort of concurrency control between the multiple transactions in IPBrick database. After a thorough analysis of the various conventional concurrency mechanisms and their incompatibility with the existing formation of IPBrick databases, it was concluded

that none of those mechanisms can produce any noticeable results in a short span of time. Consequently, an alternative approach was devised to provide the concurrency control at the abstract level for the multiple transactions of update and delete operations in an interactive environment. This scheme was developed solely for IPBrick service "User Management" as a proof of concept that was found quite efficient at the end because it was able to avoid possible data conflict in a multi-sessions based environment.

Renaming of the web pages in database was one the additional goals performed during the implementation of this thesis. Around 850 web pages were renamed in English by using a well-defined nomenclature which was helpful in overcoming the inconsistencies occurred in the names before. In spite of non-technical nature of this task, there is no denying its importance that it can be extremely helpful for customers as well as for developers too to understand the currently deployed IPBrick system. The inclusion of *Run-Time Configurations* module in the latest version of IPBrick solution was another extra goal that was attempted to be implemented for the "User Insert" service. Unfortunately due to the lack of time, it was not tested properly. But, even the attempt of its development was fully worth it because it was really very useful in understanding the practical applications of SOAP and WSDL.

Finally, all of the main goals were achieved with some decent results, as discussed in the Chapter 4, accompanied by some significantly positive ones obtained for the additional objectives. As it is said in the beginning that the refinement of the existing IPBrick solution was the main goal and that was accomplished to a very great extent at the end.

5.2 Future Work

“The only certain means of success is to render more and better service than is expected of you, no matter what your task may be.”

Og Mandino

Administrative Profiles Unit was developed with the mindset that it should be user-friendly by minimizing the required number of clicks for each operation applied on the profiles. Still, all facets of this module can be improved a little bit for a better user experience. The one of them is the further simplification of Profile Permissions Prompt. It can have an option of Select/un-Select some particular menu along with its subsequent sub-menus and pages. Other enhancement in it can be to show the previously selected Menus and Sub-Menus along with their adjacent selected web pages and also allow to select the new ones when an admin is trying to modify some profile's permissions. At the moment, the admins are allowed to select only one of the page permissions – View, Insert, Delete, Modify and all – for a web page but it can be extended further by enabling the admins to choose a combination of privileges for a web page such as Insert and Delete combination of permissions and few other possible ones. Providing another option like

select Insert/Delete/Modify option for all web pages will not be a bad facility to give, so it can be considered and implemented in future.

Ideally, Optimistic or Pessimistic or Hybrid Concurrency Control mechanisms should be used for the multi-sessions management along with building of client and server ends of IPBrick Database. This improvement will require the large investment of time along with various other resources. For the time being, the alternative of conventional Concurrency Control discussed in this thesis can be adopted and further implemented for all IPBrick entities and services which is done for *User Management* only at the moment.

Besides the actual objectives of this thesis, some additional tasks were also developed to a limited extent, their implementations can be carried forward at full scale in future. Around 850 web pages were renamed in English by using a well-defined nomenclature, the same can also be done for the rest web pages along with the appropriate changes in their system files. Such uniformity in the web pages' names can provide a better understanding of IPBrick Web Interface to its users. *Run-Time Configuration* unit was not fully implemented due to the shortage of time but it can be implemented in the IPBrick System by using its solution partially developed in this thesis. Full implementation of this feature will really boost up the execution speed of users' operations performed on IPBrick web interface. For enhancement in the error handling aspect of this feature, it is recommended that a procedure should be developed that can generate a response message with more explicit meaning – pointing towards the cause of some error/fault due to which it occurs during the execution – instead of returning a negative value. It will be helpful in aligning the admins' actions according to the operational requirements of Run-Time Configuration module.

Appendix A

Useful Code Snippets

In this chapter, author has mentioned some of the utility functions and code snippets which were found useful during the development process of thesis.

A.1 Functions for Profile Insertion

The function *reload_all()* is used to reload the given web page if the profile's name is not specified. Second function *CheckSubmit* validates the profile's name inserted by the user and then confirms the creation of profile from user by showing a JavaScript based alert. After that, it performs an action appropriate to user's response.

```
1
2 <script language="JavaScript">
3
4 function reload_all()
5 {
6     document.form_profile.pagina.value="profile_ins";
7     document.form_profile.submit ();
8 }
9
10 function CheckSubmit ()
11 {
12     //TESTE ao name
13     if (document.form_profile.name.value=="") {
14         window.alert("<?echo _("Please Specify the Name!");?>");
15         return false;
16     }
17     if (document.form_profile.name.value==".")
18     {
19         window.alert("<?echo _("Invalid Name!");?>");
20         return false;
21     }
22     sair = window.confirm("
23     <?echo _("Do you really want to create this Profile!");?>");
24     return sair;
```

```

25 }
26 </script >

```

A.2 Function for the specification of Profile Permissions

The function *plotTree* explodes a single-dimensional array of menus and sub-menus into a fully blown tree structure based on the delimiter "." found in it's key. The argument *counter* is for maintaining the values of the higher nodes(parents). Within the scope of this function another function *printPages* is called that prints the web page(s) along with its permissions(view, insert, modify, delete and all) under their respective menus or sub-menus..

```

1
2 <?
3 function plotTree($arr,$counter=0){
4   global $menu_permission,$menuNames;
5   $menu_level='';
6   if(is_array($arr) && key($arr)=="__base_val"){
7     foreach ($arr as $k=>$v){
8       $show_val = (is_array($v) ? $v["__base_val"] : $v);
9       $menu_level=array_search($show_val,$menuNames);
10
11      if ($k == "__base_val"){
12        if($counter==0){
13          echo '<fieldset class="expUnexp"><legend><a class="titulos">Menu ' .
14          $show_val.'</a></legend>';
15          echo '<div class="hidere">';
16          echo '<li><input class="checkbox" type="checkbox" id="
17          menu_permission_'.$menu_level.'" name="menu_permission['.$menu_level.]"
18          value="'.$menu_level.'"><a class="titulosb">';
19          $counter++;
20          }
21          else {
22            echo '<div class="hidere">';
23            echo '<li><input class="checkbox" type="checkbox" id="
24            menu_permission_'.$menu_level.'" name="menu_permission['.$menu_level.]"
25            value="'.$menu_level.'"><a class="titulos">';
26            }
27            echo $show_val.'</a></li>';
28            printPages($show_val);
29            echo '<ul>';
30            continue;
31          }
32        }
33        // determine the real value of this node.
34        if (is_array($v)){
35          //this is what makes it recursive, return for childs
36          plotTree($v,$counter++);
37        } else {

```

```

32         echo '<li><input class="checkbox" type="checkbox" id="
menu_permission_'. $menu_level. '" name="menu_permission['. $menu_level. ']"
value="'. $menu_level. '"><a class="titulos">'. $show_val. '</a></li>';
33         printPages($show_val);
34     }
35 }
36 echo '</ul>';
37 }
38 else{
39     $menu_level=array_search($arr, $menuNames);
40     echo '<fieldset class="expUnexp"><legend><a class="titulos">Menu '. $arr. '
</a></legend>';
41     echo '<div class="hidere">';
42     echo '<li><input class="checkbox" type="checkbox" id="menu_permission_'.
$menu_level. '" name="menu_permission['. $menu_level. ']" value="'. $menu_level
. '"><a class="titulosb">'. $arr. '</a></li>';
43     printPages($arr);
44
45 }
46 }
47
48 /*Function to subsequently Print the Web Pages Names correspondig to Menus and
Sub-Menus along with Page Permissions*/
49
50 function printPages($title){
51     global $dbsoft, $menuNames, $page_permission;
52     $menu_level=array_search($title, $menuNames);
53     $menusRec = $dbsoft->getPagesByMenu($menu_level);
54     $check=array();
55     for ($k=0; $k<count($menusRec); $k++){
56
57         $page=$dbsoft->getPageByPageId($menusRec[$k]->id_page);
58
59         if($k==0 || !in_array($page[0]->page, $check)){
60             echo '+ <a class="dados">'. $page[0]->page;
61             array_push($check, $page[0]->page);
62             echo '<div style="display:inline;" id="autoUpdate" class="autoUpdate">
';
63             echo '<input class="radioView" id="radioView" type="radio" name="
page_permission['. $menusRec[$k]->id_menu. ']" value="view">'. '<label>View</
label>';
64             echo '&nbsp;';
65             echo '<input class="radio" id="radio" type="radio" name="
page_permission['. $menusRec[$k]->id_menu. ']" value="ins">'. '<label>Insert </
label>';
66             echo '&nbsp;';
67             echo '<input class="radio" id="radio" type="radio" name="
page_permission['. $menusRec[$k]->id_menu. ']" value="mod">'. '<label>Modify
</label>';

```

```

68     echo '&nbsp;';
69     echo '<input class="radio" id="radio" type="radio"
70     name="page_permission[\' . $menusRec[$k]->id_menu . \']"
71     value="del">\' . \'<label>Delete </label>\' ;
72     echo '&nbsp;';
73     echo '<input class="radioAll" id="radioAll" type="radio" name="
page_permission[\' . $menusRec[$k]->id_menu . \']" value="all">\' . \'<label>All </
label>\' ;
74     //echo '</a><br>';
75     echo '</div></a><br>';
76     }
77
78 }
79 }
80
81 ?>

```

Jquery's functions were used for adding the following interactive features in Profile Permissions' Prompt:

- A text based click option which can easily expand/collapse the Menu tree
- All web pages are selected with View or All option if the respective radio button is clicked
- All Menus and Sub-Menus get selected or un-selected if the corresponding check-box is checked or un-checked respectively
- Only a single radio button can be selected from the set of radio buttons specified for all, view, and custom options

The code of these features are commented in the snippet given below:

```

1
2 <script type="text/javascript">
3
4 $(document).ready(function() {
5
6     //For Expanding/Collapsing Menu
7     $(' .expUnexp').click(function() {
8         $(this).find(' .hidere').toggle(true);
9     });
10
11     //$("# .expUnexp").find(" .hidere").toggle();
12     $("# .expUnexp").find(" .hidere").slideToggle();
13
14     // For the Selection of All Menus
15     $('#select_all').on('click',function(){
16         if(this.checked){
17             $(' .checkbox').each(function(){
18                 this.checked = true;

```



```

19         });
20         //$('.radio ').attr(" disabled", false);
21     } else {
22         $('.checkbox').each(function () {
23             this.checked = false;
24         });
25         //$('.radio ').attr(" disabled", true);
26     }
27 });
28
29 $('.checkbox').on('click', function () {
30     if($('.checkbox: checked').length == $('.checkbox').length) {
31         $('#select_all').prop('checked', true);
32     } else {
33         $('#select_all').prop('checked', false);
34     }
35 });
36
37 //For Selection of All option
38 $("#radio_All").click(function () {
39     if ($(this).is(':checked')) {
40         <?$pro_type=2?>
41         $("input:radio.radioAll").attr("checked", "checked");
42     }
43     else {
44         //$("#input:radio.radioView").attr("checked", "checked");
45         $("#input:radio.radioAll").removeAttr("checked");
46     }
47 });
48
49 $("#radio_All").click(function () {
50     if ($(this).is(':checked')) {
51         <?$pro_type=2?>
52         $(".radioAll").attr("checked", "checked");
53     }
54     else {
55         //$(".radioView").attr("checked", "checked");
56         $(".radioAll").removeAttr("checked");
57     }
58 });
59
60 $('.radioAll').on('click', function () {
61     if($('.radioAll: checked').length == $('.radioAll').length) {
62         <?$pro_type=2?>
63         $('#radio_All').prop('checked', true);
64     } else {
65         $('#radio_All').prop('checked', false);
66         $('#radio_Custom').prop('checked', true);
67     }
68 });

```

```

68
69 //For Selection of View option
70 $("#radio_View").click(function() {
71     if ($(this).is(':checked')){
72         <?$pro_type=1?>
73         $("input:radio.radioView").attr("checked", "checked");
74     }
75     else {
76         //$("input:radio.radioView").attr("checked", "checked");
77         $("input:radio.radioView").removeAttr("checked");
78     }
79 });
80
81 $("#radio_View").click(function() {
82     if ($(this).is(':checked')){
83         <?$pro_type=1?>
84         $(".radioView").attr("checked", "checked");
85     }
86     else {
87         //$(".radioView").attr("checked", "checked");
88         $(".radioView").removeAttr("checked");
89     }
90 });
91 $('radioView').on('click',function(){
92     if($('radioView:checked').length == $('radioView').length){
93         <?$pro_type=1?>
94         $('#radio_View').prop('checked', true);
95     } else {
96         $('#radio_View').prop('checked', false);
97         $('#radio_Custom').prop('checked', true);
98     }
99 });
100
101 //For Selection of Custom Option
102 $("#radio_Custom").click(function() {
103     if ($(this).is(':checked')){
104         $(".radioView").removeAttr("checked");
105         $(".radioAll").removeAttr("checked");
106     }
107 });
108 $('radioCustom').on('click',function(){
109     $('#radio_All').prop('checked', false);
110     $('#radio_View').prop('checked', false);
111 });
112 //For Selection of one option
113 $(".radio_View").change(function () {
114     <?$pro_type=1?>
115     $('radio_All').not(this).prop('checked', false);
116     $('radio_Custom').not(this).prop('checked', false);

```

```

117     });
118
119     $(".radio_All").change(function () {
120         <?$pro_type=2?>
121         $('radio_View').not(this).prop('checked', false);
122         $('radio_Custom').not(this).prop('checked', false);
123     });
124     $(".radio_Custom").change(function () {
125         <?$pro_type=3?>
126         $('radio_All').not(this).prop('checked', false);
127         $('radio_View').not(this).prop('checked', false);
128     });
129
130     //unchecked hide
131     $('checkbox').change(function(){
132         if(this.checked)
133             $('#autoUpdate').fadeIn('slow');
134         else
135             $('#autoUpdate').fadeOut('slow');
136
137     });
138
139 });
140
141 //Function for Expanding/Collapsing all Menus
142 function expandCollapse() {
143     if($(".hidiers").css('display') == 'none') {
144         $("#expand-collapse").html("Collapse All");
145         $(".hidiers").show("slow");
146     } else {
147         $("#expand-collapse").html("Expand All");
148         $(".hidiers").hide("slow");
149     }
150 }
151
152 </script >

```

The function *Update* is used to check the radio button checked by the admin from the ones designated for all, view and custom options. If none of them are found selected then the custom option will be used and set as a default value of the profile type.

```

1
2 <script language="javascript">
3 function Update()
4 {
5     if (document.getElementById('radio_View').checked)
6         document.form_profile_permissions.pro_type.value=1;
7     else if (document.getElementById('radio_All').checked)
8         document.form_profile_permissions.pro_type.value=2;
9     else if (document.getElementById('radio_Custom').checked)

```

```

10     document.form_profile_permissions.pro_type.value=3;
11     else
12     document.form_profile_permissions.pro_type.value=3;
13 }
14
15 </script >

```

A.3 Funtions for Profile's Allocation

First funtion *AddUserGrp* adds a user's login into an array *f_utilizador2* while the second one deletes a user's login from an array named *f_utilizador1*.

```

1
2 <script language="JavaScript">
3
4 function AddUserGrp ()
5 {
6
7     for(i=document.form_grupo_ver.f_utilizador2.length-1; i>=0; i--)
8     {
9         if (document.form_grupo_ver.f_utilizador2.options[i].selected) {
10             document.form_grupo_ver.f_utilizador1.options
11             [document.form_grupo_ver.f_utilizador1.options.length] =
12             new Option (document.form_grupo_ver.f_utilizador2.
13             options[i].text , document.form_grupo_ver
14             .f_utilizador2.options[i].value , false , false);
15             document.form_grupo_ver
16             .f_users_inser.value = document.form_grupo_ver.f_users_inser.value+'';
17             +document.form_grupo_ver.f_utilizador2.options[i].value;
18             document.form_grupo_ver.f_utilizador2.options[i] = null;
19         }
20     }
21     document.form_grupo_ver.f_submit_action.value = "11";
22     document.form_grupo_ver.submit ();
23     return ;
24 }
25
26 function DelUserGrp ()
27 {
28     for(i=document.form_grupo_ver.f_utilizador1.length-1;i>=0;i--)
29     {
30         if (document.form_grupo_ver.f_utilizador1.options[i].selected) {
31             document.form_grupo_ver.f_utilizador2.options
32             [document.form_grupo_ver.f_utilizador2.options.length] =
33             new Option (document.form_grupo_ver.f_utilizador1.
34             options[i].text , document
35             .form_grupo_ver.f_utilizador1.
36             options[i].value , false , false);

```

```

37     document.form_grupo_ver.f_users_apaga.value= document.form_grupo_ver.
38     f_users_apaga.value
39     +':'+document.form_grupo_ver.f_utilizador1.
40     options[i].value;document.form_grupo_ver.f_utilizador1.
41     options[i] = null;
42   }
43 }
44 document.form_grupo_ver.f_submit_action.value = "22";
45 document.form_grupo_ver.submit ();
46 return;
47 }
48
49 </script >

```

A.4 Funtions for applying permission on a web page

This code checks the value of input tag named *page_permission* for a given web page then hides some *anchor* tags from its display based on the wildcard IDs of their *href* attributes.

```

1
2 <script type="text/javascript">
3
4 $(document).ready(function() {
5
6   if($("#page_permission").val()=="ins"){
7     $('a[href*="_ins"]').hide(); //ins: insert
8     $('a[href*="_lic"]').hide(); //lic: license
9     $('a[href*="_actv"]').hide(); //actv: activate
10    $('a[href*="_exp"]').hide(); //exp: export
11  }
12  else if($("#page_permission").val()=="del"){
13    $('a[href*="_del"]').hide();
14    $('a[href*="_delAll"]').hide();
15  }
16  else if($("#page_permission").val()=="mod"){
17    $('a[href*="_ins"]').hide();
18    $('a[href*="_del"]').hide(); //del: delete
19    $('a[href*="_delAll"]').hide(); //delAll: delete all
20    $('a[href*="_lic"]').hide();
21    $('a[href*="_actv"]').hide();
22    $('a[href*="_ord"]').hide(); //ord: order
23    $('a[href*="_mbr"]').hide(); //mbr: member
24    $('a[href*="_excep"]').hide(); //excep: exception
25    $('a[href*="_exp"]').hide();
26    $('a[href*="_gen"]').hide(); //gen: generate
27    $('a[href*="_blk"]').hide(); //blk: block
28  }
29  else if($("#page_permission").val()=="view"){

```

```
30 $( 'a[href*="_ins"]' ).hide ();
31 $( 'a[href*="_mod"]' ).hide ();
32 $( 'a[href*="_del"]' ).hide ();
33 $( 'a[href*="_delAll"]' ).hide ();
34 $( 'a[href*="_lic"]' ).hide ();
35 $( 'a[href*="_actv"]' ).hide ();
36 $( 'a[href*="_ord"]' ).hide ();
37 $( 'a[href*="_mbr"]' ).hide ();
38 $( 'a[href*="_excep"]' ).hide ();
39 $( 'a[href*="_exp"]' ).hide ();
40 $( 'a[href*="_gen"]' ).hide ();
41 $( 'a[href*="_blk"]' ).hide ();
42 }
43 else
44 {}
45 });
46
47 </script >
```

Appendix B

Relational Algebra

B.1 Administrative Profiles Unit

B.1.1 Queries for the *profile* Table

Get <i>IDs</i> column of the valid user with 'X'	$\Pi_{idprofile}(\sigma_{iduser='X'}(valida))$
Get <i>all</i> columns of the profile with ID 'X'	$\Pi_{idprofile,profile_name,description,profile_type}(\sigma_{idprofile='X'}(profile))$
Get <i>all</i> columns of the profile with name 'XYZ'	$\Pi_{idprofile,profile_name,description,profile_type}(\sigma_{profile_name='XYZ'}(profile))$
Get the <i>IDs</i> of the unassigned profiles	$\Pi_{idprofile}(profile) - \Pi_{idprofile}(valida)$

B.1.2 Queries for the *profile_menu* Table

Get <i>all</i> columns of profile_menu for the profile ID 'X'	$\Pi_{idprofile_menu,idprofile,id_menu,menu_level}(\sigma_{idprofile='X'}(profile_menu))$
Get <i>menu_level</i> column of profile_menu for the profile ID 'X'	$\Pi_{menu_level}(\sigma_{idprofile='X'}(profile_menu))$

B.1.3 Queries for the *profile_page* Table

Get <i>all</i> columns of <i>profile_page</i> for the profile ID 'X'	$\Pi_{id_{profile_page}, id_{profile}, id_{page}, menu_level, page_permission}(\sigma_{id_{profile}='X'}(profile_page))$
Get <i>all</i> columns of <i>profile_page</i> for the page ID 'X'	$\Pi_{id_{profile_page}, id_{profile}, id_{page}, menu_level, page_permission}(\sigma_{id_{page}='X'}(profile_page))$
Get <i>all</i> column of <i>profile_page</i> for the menu_level 'X.Y'	$\Pi_{id_{profile_page}, id_{profile}, id_{page}, menu_level, page_permission}(\sigma_{menu_level='X.Y'}(profile_page))$
Get <i>page_permission</i> columns of <i>profile_page</i> for the profile ID 'X' and menu_level 'Y.Z'	$\Pi_{page_permission}(\sigma_{id_{profile}='X' \wedge menu_level='Y.Z'}(profile_page))$

B.2 Multi-Sessions Management Unit

B.2.1 Queries for the *transaction* Table

Get <i>all</i> columns of the transaction	$\Pi_{id_{transaction}, id_{servico}, id_{entity}, entity}(transaction)$
Get <i>servico</i> column of the transaction	$\Pi_{servico}(transaction)$
Get <i>servico</i> column of the transaction for the session ID 'X'	$\Pi_{servico}(\sigma_{id='X'}(transaction))$
Get columns <i>id</i> , <i>id_entity</i> and <i>entity</i> of the transaction for the service 'XYZ'	$\Pi_{id, id_entity, entity}(\sigma_{servico='XYZ'}(transaction))$
Get <i>all</i> columns of the transaction for the session ID 'X' and 'XYZ'	$\Pi_{servico}(\sigma_{id='X' \wedge servico='XYZ'}(transaction))$

B.2.2 Queries for the *sessao* Table

Get <i>all</i> columns of <i>sessao</i>	$\Pi_{id, sessid, utilizador, inicio, expira}(sessao)$
Get <i>all</i> columns of <i>sessao</i> for the user 'XYZ'	$\Pi_{id, sessid, utilizador, inicio, expira}(\sigma_{utilizador='XYZ'}(sessao))$

Appendix C

Mock-ups of web pages

It should be noted that Mock-ups of few web pages – related to Administrative Profiles Unit – are shown in this appendix which were created during the preparatory phase of this thesis. These designs were not strictly followed during the actual implementation of their respective web pages and some changes were made in them where it was considered necessary.

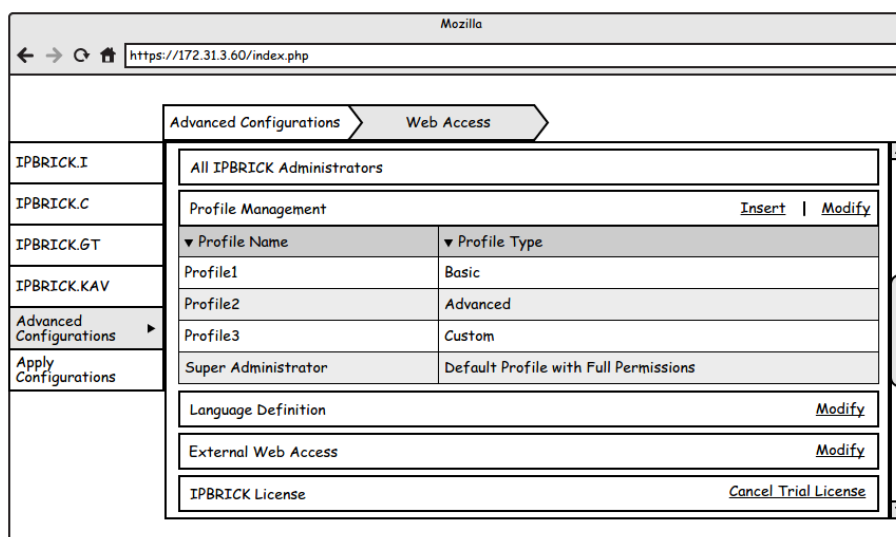
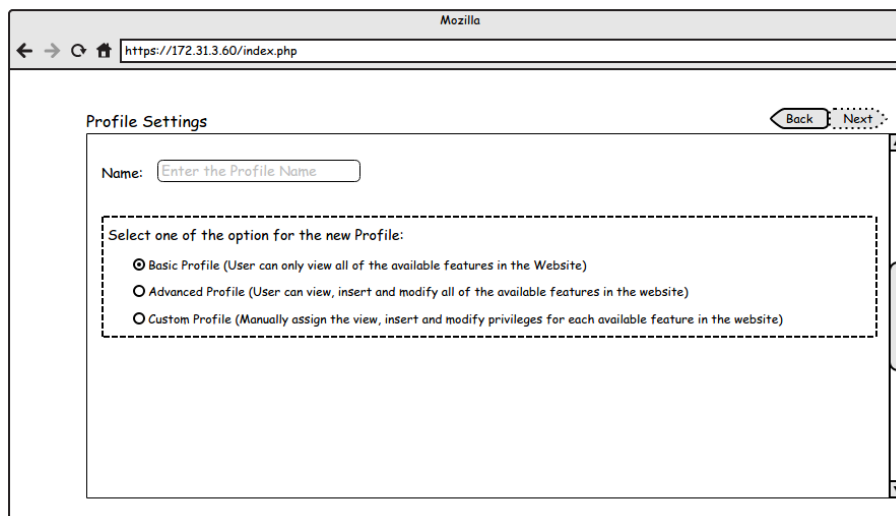


Figure C.1: Mock-up of the default web page of Administrative Profiles Unit



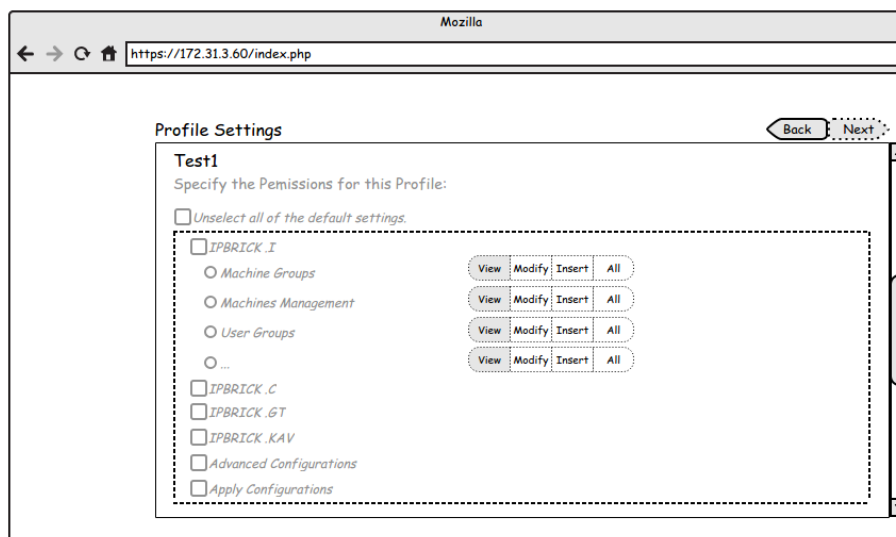
Profile Settings Back Next

Name:

Select one of the option for the new Profile:

- Basic Profile (User can only view all of the available features in the Website)
- Advanced Profile (User can view, insert and modify all of the available features in the website)
- Custom Profile (Manually assign the view, insert and modify privileges for each available feature in the website)

Figure C.2: Mock-up of the web page for Profile Insertion



Profile Settings Back Next

Test1
Specify the Permissions for this Profile:

Unselect all of the default settings.

- IPBRICK .I View Modify Insert All
- Machine Groups View Modify Insert All
- Machines Management View Modify Insert All
- User Groups View Modify Insert All
- ... View Modify Insert All
- IPBRICK .C
- IPBRICK .GT
- IPBRICK .KAV
- Advanced Configurations
- Apply Configurations

Figure C.3: Mock-up of the web page for the selection of Profile's Permissions

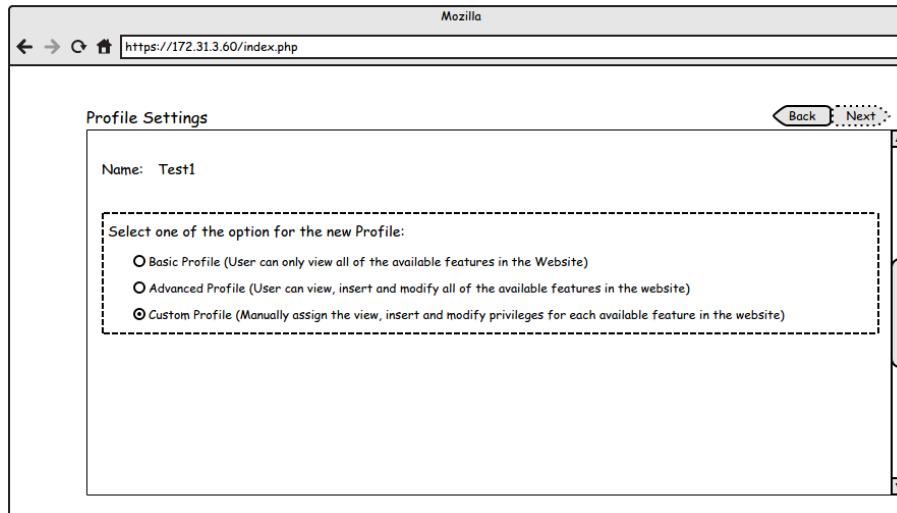


Figure C.4: Mock-up of the web page for the modification in the Profile's Definitions

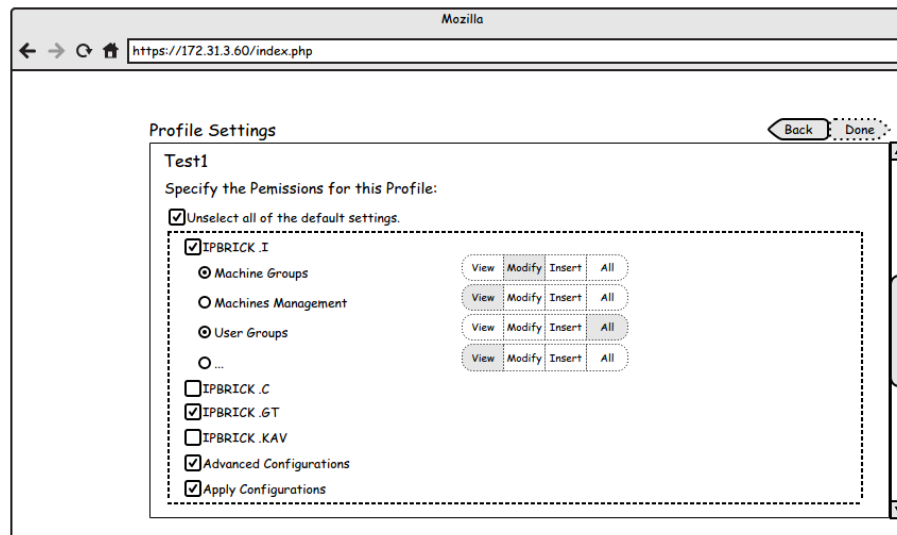


Figure C.5: Mock-up of the web page for the modification in the Profile's Permissions

Appendix D

Utility Stuff

D.1 Bash commands

ls -la directory

chmod permissions filename

chown owner filename

scp username@remotehost.edu:filename directory

php -l filename

ssh remotehost

tail -f filename

Lists all of the files and folders along with their permissions stored in a particular directory

Changes the permissions of some specific file

Changes ownership of some specific file

Copy the file from a remote host to the localhost

Checks the syntax errors in the file

Connects to a remote system

Displays the last 10 lines of the file and appends new lines to the display as new lines are added to the file

D.2 PHP Functions

basename

print_r

var_dump

error_log

Returns trailing name component of path

Prints human-readable information about an array

Dumps information about an array

Sends an error message to the defined error handling routines

D.3 Structure of the SOAP Request/Response Message

Specific elements of a SOAP Request/Response message without any payload is given below:

```
1 <?xml version="1.0" ?>
```

```

2
3 <soap:Envelope
4   xmlns:soap=" http://172.31.3.60/run-time/server/soap-envelope "
5   soap:encodingStyle=" http://172.31.3.60/run-time/server/soap-encoding ">
6   <soap:Header>
7     ...
8   </soap:Header>
9   <soap:Body>
10    ...
11    <soap:Fault>
12      ...
13    </soap:Fault>
14  </soap:Body>
15 </soap:Envelope>

```

A SOAP message has a root element *Envelope* with the namespace soap as *http://172.31.3.60/run-time/server/soap-envelope*. The *soap:encodingStyle* attribute determines the data types used in the file, but SOAP itself does not have a default encoding.

soap:Envelope is a mandatory element, but the next element *soap:Header*, is optional and usually contains information relevant to authentication and session handling. Here it should be noted that SOAP protocol does not offer any built-in authentication. Then, there is another required element *soap:Body* which contains the actual RPC message including method names and in the case of a response, the return values of the method. Last attribute *soap:Fault element*(child of *soap:Body*) is optional; if it is present then it holds any error messages or status information for the SOAP message.

D.4 Structure of WSDL

Similar to SOAP messages, WSDL has a specific schema to adhere and specific elements that must be in place to be valid

```

1
2 <definitions>
3   <types>
4     .....
5   </types>
6   <message>
7     <part></part>
8   </message>
9   <portType>
10    .....
11  </portType>
12  <binding>
13    ....
14  </binding>
15  <service>
16    ....

```



```
17 </ service >  
18 </ definitions >
```

The root element of the WSDL is the *definitions* element which provides the definition of the web service. The *types* element describes the type of data used and in the case of WSDL, XML schema is used. Within the *messages* element, is the definition of the data elements for the service. Each *messages* element can contain one or more part elements. The *portType* element defines the operations that can be performed with a web service and that are request and response messages. Within the *binding* element, contains the protocol and data format specifications for a particular *portType*. Finally, the *service element* which defines a collection of service element containing the URI (location) of a web service.

References

- [1] IPBRICK <support@ipbrick.com>. Ipbrick eshop. http://eshop.IPBrick.com/eshop/software.php?cPath=7_74, [Accessed: 05- Nov- 2015].
- [2] N.A. HTML reference. <http://www.w3schools.com/tags/>, [Accessed: 21- Feb- 2016].
- [3] A.Restivo. HTML5. <https://paginas.fe.up.pt/~arestivo/presentation/html5/#1>, [Accessed: 24- Mar- 2016].
- [4] N.A. Techwelkin. <http://techwelkin.com/difference-between-static-and-dynamic-web-pages>, [Accessed: 10- Dec- 2015].
- [5] A.Restivo. JavaScript. <https://paginas.fe.up.pt/~arestivo/presentation/javascript/#1>, [Accessed: 17- Apr- 2016].
- [6] A.Restivo. jQuery. <https://paginas.fe.up.pt/~arestivo/presentation/jquery/#1>, [Accessed: 13- Apr- 2016].
- [7] N.A. jquery. <https://jquery.com/>, [Accessed: 12- Dec- 2015].
- [8] N.A. Get smarty. http://www.smarty.net/crash_course, [Accessed: 13- Dec- 2015].
- [9] A.Restivo. PHP5. <https://paginas.fe.up.pt/~arestivo/presentation/php/#1>, [Accessed: 11- Apr- 2016].
- [10] N.A. Top ten reviews. <http://php-editor-review.toptenreviews.com/php-perl-or-python-which-should-you-use-.html>, [Accessed: 20- Dec- 2015].
- [11] J.Linstrom. Optimistic concurrency control for real-time database systems. Technical report, University of Helsinki, Finland, January 2003.
- [12] J.Linstorm. Efficient Optimistic Concurrency Control for mobile real-time transactions in a wireless data broadcast environment. <https://www.cs.helsinki.fi/u/jplindst/papers/icipca2010.pdf>, [Accessed: 15- Jan- 2016].
- [13] V.Hadzilacos P.A.Bernstein and N.Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, N.A. edition, 1987.
- [14] C.Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Inc. New York, NY, USA, N.A. edition, 1986.

- [15] J.Gray and A.Reuter. *Transaction Processing: Concepts and Techniques*. N.A., First edition, 1993.
- [16] A.Thomasian. *Database Concurrency Control*. Kluwer Academic Publishers, n.a. edition, 1996.
- [17] K.Ramamritham B.Purimetla, R.M.Sivasankaran and J.A.Stankovic. *Real-time Databases: Issues and Applications*. Prentice Hall, n.a. edition, 1996.
- [18] P.Konana I.Viguier A.Datta, S.Mukherjee and A.Bajaj. Multiclass Transaction Scheduling and Overload Management in firm real-time database systems. *N.A.*, pages 29–54, March 1996.
- [19] B.Yang G.Li and J.Chen. Efficient Optimistic Concurrency Control for mobile real-time transactions in a wireless data broadcast environment. *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 443–446, September 2005.
- [20] M.T.Ozsu and P.Valduriez. *Principles of Distributed Database System*. Prentice Hall, Second edition, 1999.
- [21] G.Sreedhar. *Design Solutions for Improving Website Quality and Effectiveness*. N.A., n.a. edition, 2016.
- [22] N.A. 15 Concurrency control. <http://www.inf.fu-berlin.de/lehre/SS05/19517-V/FolienEtc/dbs04-19-ConCtrl-1-2.pdf>, [Accessed: 14- Jan- 2016].
- [23] Rasmus Pagh. Lecture 7: Concurrency control. <http://www.itu.dk/people/pagh/DBT07/concurrency-control.pdf>, [Accessed: 25- Jan- 2016].
- [24] K.Y.Lam K.W.Lam and S.Hung. Real-time Optimistic Concurrency Control Protocol with dynamic adjustment of serialization order. *In Proceedings of the IEEE Real-Time Technology and Application Symposium*, pages 174–179, May 1995.
- [25] R.A.Lorie K.P.Eswaran, J.N.Gray and I.L.Traiger. The notions of Consistency and Predicate Locks in a database system. *Communications of the ACM*, pages 624–633, November 1976.
- [26] H.T.Kung and J.T.Robinson. On Optimistic methods for Concurrency Control. *ACM Transactions on Database Systems*, pages 213–226, June 1981.
- [27] R.Bouaziz A.Makni and F.Gargouri. Performance evaluation of an Optimistic Concurrency Control algorithm ensuring strong consistency for transaction time relations. *In Enterprise Information Systems and Web Technologies*, pages 258–265, January 2007.
- [28] S.L.Hung K.W.Lam, V.Lee and K.Y.Lam. An augmented priority ceiling protocol for hard real-time systems. *journal of computing and information. Proceedings of Eighth International Conference of Computing and Information*, pages 894–866, June 1996.
- [29] N.A. Postgresql anti-patterns: Read-Modify-Write cycles. <http://blog.2ndquadrant.com/postgresql-anti-patterns-read-modify-write-cycles/>, [Accessed: 22- Mar- 2016].
- [30] G.Thornton. N.a. <http://www.orafaq.com/papers/locking.pdf>, [Accessed: 07- Apr- 2016].

- [31] S.Thorpe. RSS. <https://www.sitepoint.com/web-services-with-php-and-soap-1/>, [Accessed: 10- May- 2016].
- [32] S.Thorpe. RSS. <https://www.sitepoint.com/web-services-with-php-and-soap-2/>, [Accessed: 10- May- 2016].
- [33] N.A. Chapter 1 – fundamentals of web application performance testing. <https://msdn.microsoft.com/en-us/library/bb924356.aspx>, [Accessed: 25- Jan- 2016].
- [34] N.A. Chapter 2 – fundamentals of web application performance testing. <https://msdn.microsoft.com/en-us/library/bb924356.aspx>, [Accessed: 23- Jan- 2016].
- [35] N.A. Chapter 14 – fundamentals of web application performance testing. <https://msdn.microsoft.com/en-us/library/bb924356.aspx>, [Accessed: 23- Jan- 2016].