Study of LIKIR and Licha; New Paradigm of Version Control System Using LIKIR

Asim Shahzad
June, 2010

Master of Science Thesis
Stockholm, Sweden 2010

TRITA-ICT-EX-2010:168

Department of Information Technology
School of Information and Communication Technology
Royal Institute of Technology(KTH)
Stockholm, Sweden

Master Thesis Project

Study of LIKIR and Licha; New Paradigm of Version Control System Using LIKIR

Author:
Asim Shahzad

Supervisor:
Mr. Renato Lo Cigno
Examiner:
Mr. Markus Hidell

June 2010

Abstract

Peer-to-peer(p2p) systems, compared to conventional client-server systems are getting the attention now a days. P2p systems demonstrate valuable benefits such as resiliency, high scalability and efficiency, and robustness (good resistance against random node failures). Structured P2P systems use distributed hash tables (DHTs) to index and search content and resources in an overlay network. Kademlia is one of the DHT among many DHTs. With the advent of DHTs finding nodes in a p2p environment has gotten better. The special property of Kademlia is that it uses novel XOR metric approach to measure the distance among the points (peers/clients). Most of Kademlia's advantages depend on the usage of novel XOR metric. In this thesis, we will concentrate on the discussion, analysis and evaluation of LIKIR (a secure P2P framework on top of Kademlia) and P2P applications that can be built on top of LIKIR. One of these applications is LIKIR chat (LiCha) which is a prototype chat program. It is already available and presents a good starting point to understand the architecture and potential strength of such systems. Our work also deals with understanding LiCha, adding new functionality in LiCha. Finally, investigating the possibility of P2P version control system, as well as describing the basic protocol primitives and implementation details of P2P version control system, which can be a extremely innovative application for P2P systems.

Sammanfattning

Peer-to-peer (P2P) system, jämfört med konventionella klient-server-system får den uppmärksamhet nu för tiden. P2p-system visar värdefulla förmåner såsom stabilitetsskäl, hög skalbarhet och effektivitet och robusthet (god motståndskraft mot slumpmässiga nod fel). Strukturerade P2P system använder distribuerade hashtabeller (DHTs) att indexera och söka efter innehåll och resurser i ett överlappande nät. Kademlia är en av DHT bland många DHTs. Med intåget av DHTs hitta noder i ett P2P miljö har blivit bättre. Den speciella egendom Kademlia är att den använder nya XOR metriska metod för att mäta avståndet mellan punkterna (kamrater / klienter). De flesta av Kademlia fördelar beroende påanvändningen av nya XOR metriska. I denna avhandling kommer vi att koncentrera oss pådiskussion, analys och utvärdering av LIKIR (en säker P2P ram ovanpåKademlia) och P2P-applikationer som kan byggas ovanpåLIKIR. En av dessa applikationer är LIKIR chat (LiCha) som är en prototyp chatt program. Det finns redan tillgänglig och presenterar en bra startpunkt för att förståarkitektur och potential för sådana system. Vårt arbete handlar ocksåom förståelse LiCha, lägga till ny funktionalitet i LiCha. Slutligen undersöker möjligheten att P2P versionshanteringssystem, samt beskriver de grundläggande protokollet primitiva och införandet av P2P versionshanteringssystem, vilket kan vara ett mycket innovativt ansökan om P2P-system.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Mr. Renato lo Cigno - Associate Professor (University of Trento) who always steered me through every phase of my research work. Without his supervision and anticipation it would have been difficult for me to finalize my research work. His supervision approach put me in a position where I could understand things much systematically and his critical yet fruitful remarks allowed me to dig deeper into the subject matter. I am also very thankful to Mr. Renato and University of Trento for making my stay pleasant.

Special thanks to Mr. Giancarlo Ruffo - Associate Professor and Mr. Luca Aiello (University of Torino) who assisted me to understand the LIKIR and LiCha in detail. I really appreciate the help that had been provided to me throughout the development phase by Mr. Luca Aiello.

I would also like to express my gratitude to my internal supervisor Mr. Markus Hidell - Assistant Professor who gave me the opportunity to work with him and learn useful knowledge.

Finally, I am indebt to my parents and my friends for their moral, social, ethical and financial support throughout my studies, without which, I might not been able to taste the leverage of success and this research work in particular.

Contents

List of Figures

List of Tables

Abbreviations

LIKIR **c**hat	**LiCha**
Layered Identity-based **K**ademlia-like **I**nfrastructure	**LIKIR**
Peer-to-**p**eer	**p2p**
Distributed **h**ash **t**able	**DHT**
Distributed **h**ash **t**ables	**DHTs**
Personal **C**omputer	**PC**
Concurrent **v**ersion **s**ystems	**CVS**
Revision **c**ontrol **S**ystem	**RCS**
Exclusive **o**r	**XOR**
Remote **p**rocedure **c**alls	**RPC**
Distributed **d**enial of **s**ervice	**DDoS**
Identity **b**ased **s**cheme	**IBS**
Certification **s**ervice	**CS**
User **i**nformation **m**odule	**UIM**
Local **a**rea **n**etwork	**Lan**
Wide **a**rea **n**etwork	**Wan**

Chapter 1

Introduction

Peer-to-peer network protocols are the potential candidates for future internet applications due to their apparent advantages (availability, scalability, resilience to failure, simple, no bottleneck, fault tolerant etc.) over client-server architecture. More recently, their development and application has witnessed a marked increase. At present, peer-to-peer applications are one of most popular applications on the Internet along with client-server applications. Now a days, peer-to-peer applications generate a large amount of traffic on the Internet. Looking at the increase usage of P2P applications, there is a good chance that in future most of the users will shift from the client-server architecture to peer-to-peer architecture. Mostly, at present client-server architecture is being used. In client-server architecture all the resources (files, services etc.) are available on a particular system called server and all other systems who use these resources are called clients. Clients send their requests to the server for a particular resource or a service and then server responds to each and every request that comes accordingly. There are many examples of applications that use client-server architecture e.g. web servers, mail servers, ftp servers, file servers, print servers etc. But in case of peer-to-peer networks, there is no particular server. Every node can acts as both client and server. In peer-to-peer networks all the nodes are equal and they share their resources that include processing power, storage, memory etc. with one another on the network. we will see more detail regarding peer-to-peer architecture in coming chapter.

LIKIR [1] is a peer-to-peer framework that is built on top of Kademlia [2]. Kademlia is one of the many other available distributed hash tables. Kademlia has many advantages over other available DHTs. It uses novel XOR metric to locate nodes and route queries. It minimizes the amount of configuration messages among the nodes to learn about one another etc. All the peer-to-peer systems specifically those who use distributed hash tables to locate nodes and route queries suffer from one common problem that is related to the security of peer-to-peer systems. Particularly related to node identification and authorization. In peer-to-peer systems, nodes cannot be identified as they don't have certified node identities. Normally node identities are randomly generated and in node authorization, there is no proper mechanism to authorize a node on the basis of its credentials e.g. login, password etc. Kademlia also suffers from identical security problems. LIKIR remove security problems by adding security to the Kademlia protocol. It also had made changes in the configuration messages to make the protocol more secure. More specifically, LIKIR introduces an identity based verification scheme and a secure communication protocol that remove the security problems that are related with the identity verification and authorized access control. It also helps to remove and minimize a number of security threats like sybil attack, route poison attack, eclipse attack, DDoS attacks etc. These attacks can really decline the performance of a peer-to-peer network to a greater extent.

LIKIR is a newly built DHT protocol and it provides defense against well known threats. As this protocol provides identity based scheme and secure communication, it opens up opportunities for new applications that can be built on top of LIKIR due to its security and interoperability.

The goals of this thesis work include, the analysis of LIKIR [1], LiCha [3] and implementation of combined chat module in LiCha. Finally, investigating the possibility of the peer-to-peer version control system and also defining the protocol specifications and implementation details of peer-to-peer version control system.

LiCha, basically is a peer-to-peer chat program that is built on top of LIKIR and can be studied to understand the architecture of LIKIR. Mainly, LiCha will be used to analyze different parts of LiCha, processes in LiCha and how LiCha can help us design an application on top of LIKIR. LiCha is not much different from other chat applications except it only provides basic chat operations right now. Currently, LiCha has one-to-one chat functionality including with other functionalities like online contact alerts, off-line contact alerts and geographical user interface color change.

In the analysis phase both LIKIR and LiCha are analyzed to gather enough knowledge about the protocol architecture and how it works in general. During the analysis of LIKIR, we will basically study LIKIR in detail and then propose some modifications in it. Also during the analysis of LiCha, we will study LiCha and propose some modifications in it. One of the proposed modification will be developed to enhance the functionality of LiCha.

LIKIR consists of several messages that are used among the nodes to execute different commands. Newly joined peers have to follow the communication protocol and incoming, outgoing messages to communicate effectively with one another. LIKIR consists of six steps that include, initialization, join, interaction among the nodes, content storage system, bootstrap list construction and identity based signature.

At last, we will study the existing version control systems including centralized, distributed and peer-to-peer version control systems.Version control systems are used to manage documents and source code. They are very beneficiary for software development industries as it helps them to increase their productivity. We will investigate the possibility of the peer-to-peer version control system on top of LIKIR. At last, we will define the protocol specifications (primitives) and implementation details that how the system will work and can be implemented.

1.1 Objectives

This thesis work is carried out in two main parts that are related to one another in the aspect of DHT. The first part consists of an analysis of LIKIR and LiCha in detail and afterwards advising specific modification in LIKIR and LiCha. At the end of this part a combined chat module will be implemented in LiCha. In the second part, we will study the existing version control systems that include client-server, distributed and peer-to-peer version control systems. Then we will investigate and propose the functional specification and the implementation details of p2p version control system on top of LIKIR. In functional specifications, we will describe steps that show how the peers can communicate with one another, version control management and data structure specific to the version control system. These functional specifications can be used as a starting point for developing a p2p version control system. In implementation details, we will describes the details that how p2p version control system can be implemented in future.

1.2 Structure of the Report

The report is organized as follows:

Chapter 2 provides the reader with a background and related study regarding LIKIR and P2P version control system.

Chapter 3 provides a detailed analysis of LIKIR, LiCha and afterwards the implementation of the of combined chat module in LiCha.

Chapter 4 provides a detailed functional specifications of a p2p based version control system.

Chapter 5 provides the conclusion and future work of this report.

Chapter 2

Background

This chapter deals with the background study and the related study that is necessary to understand the different concepts described in this thesis report. In this chapter, we will discuss in detail about p2p networks, traditional version control systems, Kademlia and existing models of peer-to-peer version control systems. In p2p networks, we will discuss different types of p2p networks that includes structured p2p networks and unstructured p2p networks and in traditional version control system, we will discuss different types of traditionally available version control systems. Kademlia will be discussed in detail, as it is used as a core protocol to develop LIKIR. Finally, we will describe the existing p2p version control systems [4] [5]. Later, these p2p version control systems can help us to understand the required functional specifications of the proposed p2p version control system on top of LIKIR. Available p2p version control systems are at very basic level and can be used as a starting point to get an idea to create a professional or enterprise level p2p version control systems.

2.1 Peer to peer network

The peer to peer network can be defined as follows:"A distributed network architecture may be called a P2p (P-to-P, P2P) network, if the participants share a part of their own hardware resources (processing power, storage capacity, network link capacity, printers.). These shared resources are necessary to provide the Service and content offered by the network (e.g. file sharing or shared workspaces for collaboration). They are accessible by other peers directly, without passing intermediary entities. The participants of such a network are thus resource (Service and content) providers as well as resource (Service and content) requestors (Servent-concept) "[6]. In p2p network every peer is equal, which means that there is no specific client and server. Each peer can act as both client and server. All peers share their resources with each other and normally p2p networks create an overlay network which is usually used for indexing, node lookups etc.

"The internet as originally conceived in late 1960's was a peer-to-peer system. The goal of the original ARPANET was to share computing resources around United States"[7]. Advanced Research Project Agency Network (ARPANET) was the first packet switching network that was developed by United States department of defense and was ancestor of today's internet. In start ARPANET consists of UCLA (University of California, Los Angles), SRI (Stanford Research Institute), UCSB (University of California, Santa Barbara) and University of Utah and they were connected with each other on the basis of equal computing peers not as a master-slave or client-server relations. Since then much effort had been done in client-server architecture to make it perfect and currently deployed at a large scale worldwide. Currently, there has been a lot of research going on to make peer-to-peer systems mature and up to the industry standards. At present p2p systems are very famous and being used in many applications specifically files sharing, files storage etc. Currently, p2p applications generates more traffic then other applications on the internet.

A study conducted by IPOQUE [8] in the year 2007[9] and 2008/2009[10] shows the internet traffic statistics of the world. This study shows a significant amount of P2P traffic that depicts the increase usage of p2p applications.

In 2007, worlds five regions were selected to see the traffic of p2p applications, this study also includes the usage of internet telephony, video streaming, instant messaging, Skype[53], file hosting and encrypted p2p protocols. This study shows that approximately 73.79 % of the internet traffic in 2007 was used by p2p applications and the rest is used by other internet applications that include HTTP, streaming, VoIP/Skype, FTP, mail etc. as shown in the figure 2.1.

In 2008/2009, IPOQUE included more regions to study the usage of internet traffic. Regions increased up to eight which include southern Africa, Southern America, Eastern Europe, Northern Africa, Germany, Southern Europe, Middle East, and Southwest Europe. The study showed that the p2p traffic is approximately 56 % on average in these regions and still it shows increased usage of p2p applications than other available applications. The detail usage of p2p traffic can be seen as shown in the figure 2.2.

P2p networks can be divided as under[54],

- Structured peer-to-peer networks
- Unstructured peer-to-peer networks

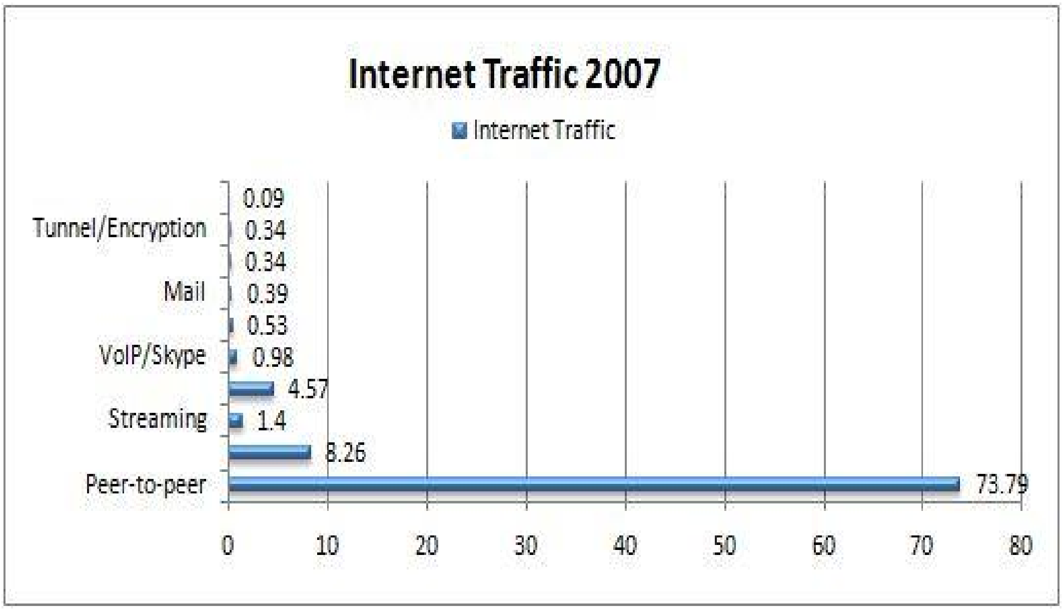


FIGURE 2.1: Internet study 2007 (Adapted from IPOQUE internet analysis 2007)[9]

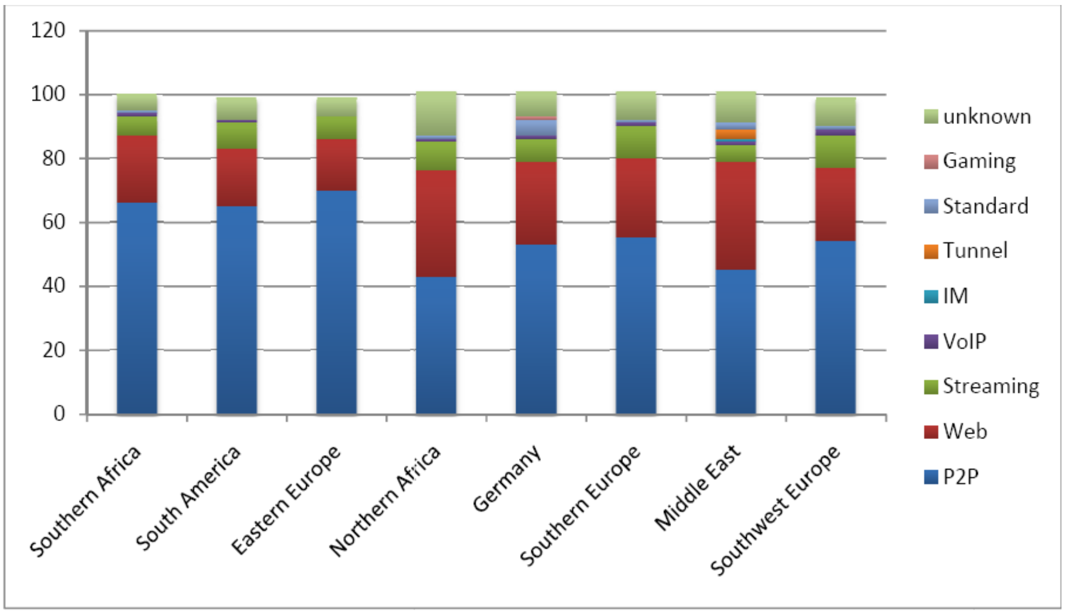


FIGURE 2.2: Internet study 2008/09 (Adapted from IPOQUE internet analysis 2008/09)[10]

2.1.1 Structured Peer-to-peer systems

Structured p2p systems normally use DHTs for indexing, node lookups etc. In structured p2p systems each peer is equal to other peers on the network. There is no central authority in p2p systems each peer can search any other peer just by sending the search query to one of the participant nodes. Structured p2p systems have the proper correlation of content and the peer who is holding that content. Structured p2p systems usually have binary tree structure and binary tree structure itself has many advantages like efficient searching, sorting etc. Structured p2p system can be seen as shown in the figure 2.3. The result will be positive all time when the query is sent to nearby nodes in order to find the nodes that have the particular content, unlike unstructured p2p systems. Popular structured peer-to-peer systems are KADEMLIA [1], CHORD [14], PASTRY [15], TAPESTRY [16] etc. we will explain briefly a couple of the structured p2p systems like chord and pastry.

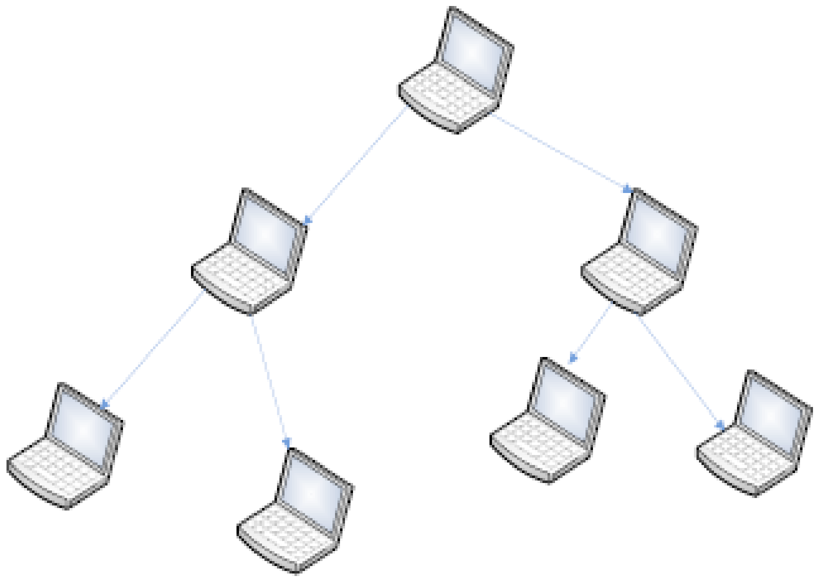


FIGURE 2.3: Structured P2p network

2.1.1.1 Chord:

Chord was developed in Massachusetts institute of technology (MIT) and is a distributed lookup protocol. It is completely decentralized and symmetric. Chord is very responsive in the event of node failure and node re-joins. A node can be found in chord network by only sending log (N) messages where N is the number of nodes. Chord only supports one function which is to find a node on the basis of a given key. It uses the consistent hash function to assigns keys to Chord nodes. Previously, it was assumed that nodes were aware about most of the other nodes in the system but in chord a node only need to know about a few nodes. Due to this reason, the routing table is distributed among the nodes.

Consistent hashing:

As discussed above, consistent hashing is used to assign keys to the chord nodes. Consistent hashing provides many advantages like load balancing, i.e. each node in chord network gets roughly the same number of keys and it also involves less number of keys when the node joins or leaves the network. Consistent hashing use SHA-1 as base hashing function and it generates m-bit identifier. Each node has its own successor and predecessor and node Keys are arranged in an identifier circle. The successor of a node can be found going clockwise in the identifier circle, and predecessor of a node can be found going anti-clockwise in the identifier circle.

One basic operation of consistent hash function is to let nodes enter and leave the network with minimal disruption. When a node A enter into the network, it gets a specified number of keys from its successor that were previously assigned to the successor of node A and when Node A leaves the network the keys assigned to Node A are re-assigned to node A's successor.

2.1.1.2 Pastry:

Pastry is another p2p system among other currently available p2p systems. Pastry creates an overlay network and provides scalability, object lookup and routing solution for p2p systems. Pastry is very similar to the Chord, it also has ring of nodes that keeps track of the keys that are assigned to particular objects. Pastry uses scalar proximity matrix to minimize the distance traveled by the messages. Pastry has been used in applications like PAST [11, 12], a global persistent storage utility and SCRIBE [13], scalable publish/subscribe system. There are many applications that are under development. Pastry can be used in many applications like group communication, file sharing, file storage and naming systems.

Pastry Design:

When a Pastry node is initialized, it is assigned a 128-bit node identifier. As it also manages the keys in a circular fashion, so node ID in this case is used to identify nodes location in the circle. Node ID is generated randomly and it can be the cryptographic hash of node's IP address or its public key.

Pastry node State:

There are certain components in each pastry node and are necessary for the establishment of better overlay network. Each node has the responsibility to manage these components properly and these components are routing table, neighborhood set and leaf set. Routing table is constructed with the help of other nodes. Neighborhood set is a collection of nodes that are closest to the local node and it is used normally to maintain locality properties. Leaf set is a collection of nodes that are closest to the local node in each direction and are used when message is routed.

Pastry Routing:

When a message comes at a node then it examines its leaf set, and if the node is found in it, then the message is sent directly to the node. Otherwise routing table is used to forward message to its correct destination.

Pastry Self-organization and adaptation:

Managing continues arrival and departure of nodes is always very important and is handled very efficiently in Pastry as well like in other DHTs.

When a new node arrives on the network, it initially sends a join request to a node. After the request is executed successfully, the node joins the network. After joining the network node has to create its routing table, neighborhood set and leaf set nodes. At the end node transmits its states to all other nodes that are in routing table, neighborhood set and leaf set.

A node leaves the network, maybe due to node failure or intentionally leaving the network. A node can be considered failed if their immediate nodes cannot communicate with the node and when it happens, the immediate nodes contact to the node and update their neighborhood set if the node is not available any more. Also if the node leaves the network, in this regard each node periodically sees that the nodes in their neighbor set are alive. If they are alive then they remains in the neighbor set otherwise they will update their immediate nodes.

2.1.2 Unstructured Peer-to-peer Systems

In unstructured p2p systems, usually there are no correlation among the content and the peers who are responsible for the data. In unstructured p2p systems queries are flooded on the network due to no perfect correlation; it is obvious that sometimes resulting peer don't have the required content that it supposed to have. More over due to query flooding there are a numbers of connections that introduces a lot of overhead onto the network. Unstructured p2p systems can be seen as shown in the figure 2.4.

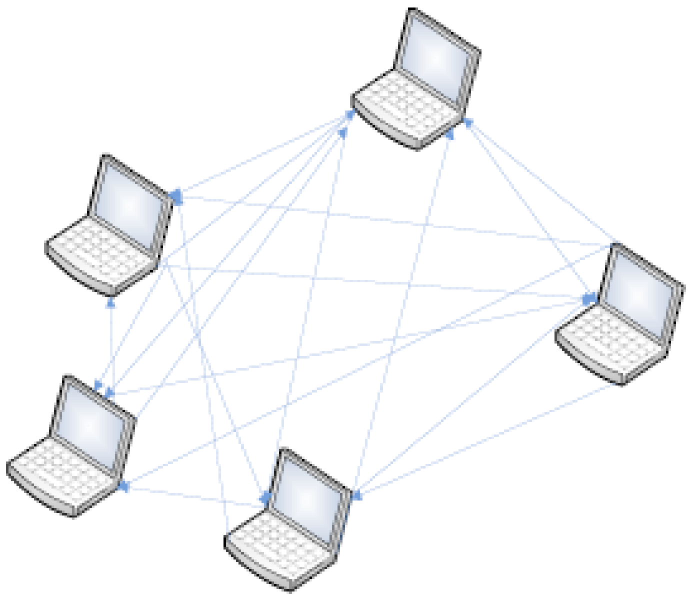


Figure 2.4: Unstructured P2p network

A very good example of unstructured p2p system can be super peers. "A super-peer is a node in a peer-to-peer network that operates both as a server to a set of clients, and as an equal in a network of super-peers"[55]. A super node act as a mini server that manages a small number of clients.Normally the super peers are better than clients due to high resources and bandwidth.

super peers are used to solve the scaling problems into unstructured p2p systems. Popular examples of unstructured p2p systems are Gnutella [17], eMule [18] etc.

2.2 Traditional version control systems

To propose functional specifications of p2p version control system. First, we need to study the background of traditional version control systems. Traditional version control systems are the basis of modern available version control systems. Traditional version control systems can be divided into two major categories.

- Centralized version control systems
- Distributed version control systems

2.2.1 Centralized version controls systems

As the name suggests this type version control system uses the client-server paradigm. Client-server paradigm used in most of the available version control. Different clients use different softwares to interact with the repository. They check out files/project and then work on them and afterwards send them back to server.The server then makes a new version of files/project and then store new version onto the server along with the old versions of the same documents. All versioned files are stored at server. This approach is very good for on site development where the clients are situated near the server. There are many examples of centralized version control systems that include.

- Concurrent versions systems[19]
- Revision control System[20]
- Subversion[21]
- MS visual source safe [22] etc.

Concurrent version systems (CVS) had been the de-facto standard for the version control industry for a long time. It was relatively reliable for software development at that time and was used by many companies like KDE e.V.[23], The GNOME Project[24], and Mozilla Foundation [25]. However, now everyone is shifting from CVS to other more flexible version control systems that can help developers to increase their productivity.

Revision control System (RCS) is one of the oldest version control systems. It automates the storing, retrieval, logging and identification of revisions, and it provides selection mechanisms for composing configurations [26]. RCS don't store the whole file, instead it only stores the delta (difference) of the file on the server which increases the performance of the system and also increases the storage capacity as compared to other available version control systems.

Subversion is now one of the most used version control system. Many projects like Apache [27], KDE [23], Free Pascal [28], FreeBSD [29], GCC [30], Python,[31] ,Google code [32], Django [33] , Ruby [34] etc. used Subversion for their software development. Most of the CVS users are shifting to Subversion because it provides many useful improvements over CVS that includes copying and renaming of files and directories, truly atomic commits, efficient handling of binary files, and the ability to be networked over HTTP and also over HTTPS.

MS visual source safe Visual SourceSafe is a version control system that protects users from file loss. It allow users to track previous versions of a file/project. It furthermore allows branching, sharing, merging, and management of file releases like other open source version control systems. It is proprietary software and is not used much as others(Subversion, CVS) are being used more frequently.

Disadvantages of Centralized version Control Systems:

- Not suitable for audio, Video and image files.
- Single point of failure
- Bottleneck if more than one user is accessing the same content.
- Expensive due to bandwidth requirement.
- Impossible to edit file without server communication.
- Increase the repository size in many cases.
- There are no off-line commits.
- They are slow.

2.2.2 Distributed version control systems

In distributed version control systems, the repository itself is replicated among the peers connected to the system. Changes made to an artifact are sent to all the other repositories outside, and this is done by a function called "push". Distributed version control systems have another function, which is called"pull". In pull function changes made by other developers are integrated into the local repositories.

Distributed version control systems remove most of the problems faced by centralized version control systems, but they introduced additional problems related to their architecture like slow, high bandwidth requirement etc. There are many distributed version control systems available, but we will briefly discuss the prominent ones that include:

- GIT [35]
- GNU Arch [36]
- Monotone [37]
- Mercurial [38] etc.

GIT is a free distributed version control system originally developed for Linux by Linus Torvalds. In GIT every repository is a full-fledged repository containing all the revisions and history regarding the software artifacts. Many high profiled software projects like Linux Kernel, WINE [39], Fedora [40]etc. uses GIT. It uses pull and push features in order to integrate the repositories.

GNU Arch is a revision control system just like GIT, CVS, subversion etc. GNU arch supports versioned renames of files and directories. GNU Arc is also widely used because of ease to use, atomic updates, intelligent merging etc. GNU Arch is good but not perfect, it often has problems at windows OS.

Monotone is not much different from other distributed version control systems. In Monotone each change set is forwarded to certain depot, and which can be CGI script, an NNTP receiver, or SMTP. From there each peer "Pull" the updates to the local repository. It also uses the encryption to provide secure communication.

Mercurial was initially built to run on Linux. It is basically used by the larger projects, and it is very fast, scalable and much easier than other version control systems. It can handle both plain text and binary files intelligently and has advanced branching and merging capabilities.

Disadvantages of distributed version control system:

- Requires a lot of bandwidth.
- Slow when many users are performing pull and push function
- merging is not always successful user has to modify the document to resolve the conflict

2.3 Study of Kademlia & existing design approaches of p2p version control system

2.3.1 Kademlia

Kademlia is a p2p distributed hash table and is described in detail in [2]. In this section our discussion will be based on [2] and [1]. Kademlia is getting famous among other distributed hash tables due to its quick search. Kademlia uses XOR metric for node lookups and to route messages between the nodes. Kademlia nodes create an overlay network where they can communicate and can share their resources with one another. Node lookup queries are routed to a set of nodes selected from the list of certain users which is maintained at each node. This list is called bucket list and will be explained soon. Kademlia uses several remote procedure (RPC) calls to execute different queries.

2.3.1.1 Kademlia Node

Kademlia node is any PC running Kademlia protocol. Each Kademlia node is assigned a random node ID. Node ID is used to identify the specific node in the network. Each node in Kademlia overlay network stores the information about the other nodes in the network that are at distance 2i and 2i+1. Node information is stored in a list which is called bucket list. Bucket list is updated after regular intervals and the nodes that are not available on the network anymore are removed from the list and new nodes are added at the tail of the list.

Kademlia node can receive different messages from other nodes and these messages can be of storing content, finding a node or finding any content on the network etc. When a Kademlia node receives a message either it is a request or reply message. The node, at first checks the node information of sender from its own bucket list who sent the message. If the node is found from the list then the node is moved to the tail of the list. If the node is not available in the list then it is added at the tail.

2.3.1.2 XOR metric

XOR means exclusive disjunction and it is also called exclusive or. Exclusive disjunction only operate at two operands if one of the operands is true then its results is true otherwise in any case its results will be false. In Kademlia, XOR metric means the bitwise exclusive disjunction of two node ids of Kademlia nodes or a node ID and a key (which is used to search the content). Both the node and the key have the same size in length and it is 160-bit. For instance if we have two 7 bit long node ids where X=1100100 and Y=1011100 then the distance between X and

Y will be D=111000 where D is the distance between X and Y. If we compare Kademlia with other popular p2p systems like Chord which manages the keys in circular form but in Kademlia the keys are managed in a unidirectional pattern. Unidirectional ensures that all requests for a given key X are routed in the same way. Kademlia XOR metric topology is also symmetric which has not been seen in other DHTs like Chord and Pastry e.g. d(x, y) = d(y, x).

2.3.1.3 Kademlia Node Lookup

In Kademlia nodes are treated as binary trees and each binary tree is further subdivided into sub-trees. These sub-trees are truly helpful for searching the nodes. Each Kademlia node can have many sub-trees that don't contain the node itself. The node should have at least one contact in each of its sub-trees. For a node, a tree division is done accordingly, highest tree is the 50% of the node contacts and then second tree is the 50% of the rest of the nodes and goes on until no node left and is shown in the figure 2.5.

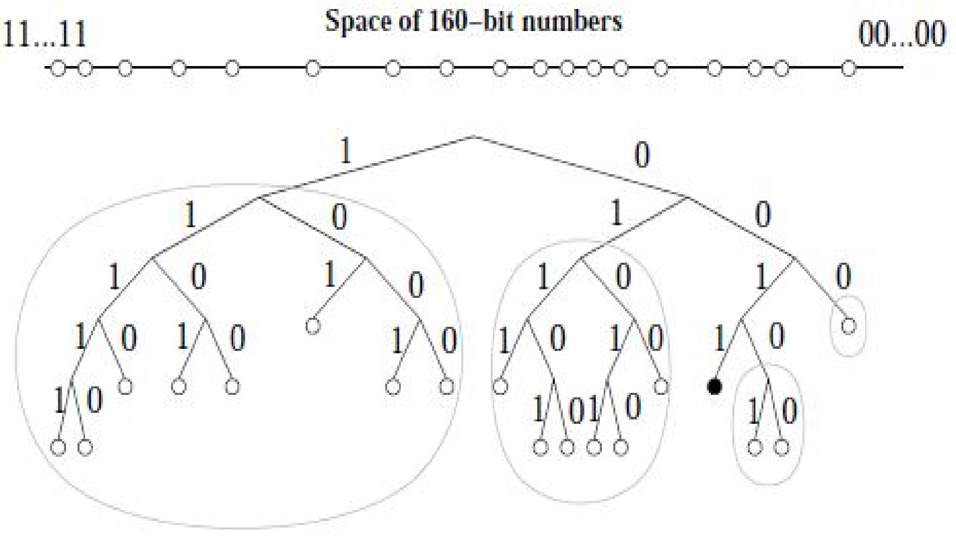


Figure 2.5: Kademlia Node lookup (Adapted from [1] node lookup)

In order to find a key the node first has to identify the proper sub-tree that contains the node based on the start of the key. A recursive node find RPC is sent to all the contacts and then these contacts return the K closest node to the required key. Node finding RPC will be discussed soon along with other RPC's and is called Find_Node RPC.

2.3.1.4 Kademlia RPC's

The Idea of remote procedure call is very important and quite simple in distributed environment. This idea is based on the thought of normal procedure in which transfer of control and data is passed to another program running on the same machine. So in remote procedure call the data and control of program is passed to another machine either in the same network or across the network. Kademlia uses four remote procedure calls that includes,

- Ping(ID)
- Store(Key, Value)
- Find_node(key)
- Find_value(key)

Ping:

Ping is a mechanism that is used to identify the reachability of any specific host on the network or outside the network. In Kademlia the ping RPC can be initiated after a specific time period as mentioned in the settings. The ping time can be changed according to the specific needs. In this RPC one peer send ping message to another peer on the basis of nodeID and other peer respond to this ping message by sending a pong message. If the peer sends the pong message which means the peer is alive and reachable otherwise the peer is not available. After both outputs bucket list is updated.

Store:

The store RPC is used to store content to a number of particular peers on the network and the content is ¡key, value¿ pair. Later, that ¡key, pair¿ content is retrieved using the key that is related to that content. These peers are the k closest peers to the key and afterwards, the store RPC is sent to them.

Find_Node:

This RPC is used to search nodes throughout the network and for that purpose a 160 bit long key is used to identify k-closest nodes to the supplied key. In response to this RPC, the recipient returns up to K triples which are ¡IP address, Port, nodeID¿ of the nodes that are closest to the key.

Find_Value:

In this RPC, the peer uses 160 bit long key to get the required content that is stored on to a particular peer. When no data is found using this RPC then this RPC act as a Find_ Node RPC and returns a set of k-closest triple to the key.

2.3.1.5 Kademlia Routing

Kademlia routing consists of routing tables that are available on every node. These routing tables include a list of nodeID's and are called buckets. The data in the buckets is very important to locate the nodes and typically consists of nodeID, IP address and port number. Every node maintains the bucket list and checks the list entries after a regular interval. If the nodes are not available then it will perform the appropriate actions onto the bucket list to make it up to date according to the current network status.

2.3.1.6 Attacks on Kademlia

Luca Maria Aiello, Marco Milanesio, Giancarlo Ruffo, Rossano Schifanella in [2] describe all the possible attacks at Kademlia and also they describe that how LIKIR can help to defend against these attacks. In this section we will briefly describe the possible attacks on Kademlia and how LIKIR can restrict all these effectively.

Routing Attacks:

In route poising a malicious node can insert wrong information into the routing table of a particular targeted peer that can be favorable to him. Kademlia faces this attack due to lack of authentication of the node credentials.

In LIKIR this problem is solved with the help of combined usage of AuthIdA and AuthAB (where A and B are two particular nodes) which will be explained in coming chapters. Every node has to show these credentials before inserting anything into the routing table. If somehow the malicious node inserted the entries then he also has to prove him/herself before doing that. Node will also show his/her credentials so this way the node can be tracked and afterwards it can be black listed due to illegal activity. It is not easy to forge nodeID because it is randomly generated by the CS (certification service).

Sybil attack:

Sybil attack is an attack where a number of nodes are initiated under a single entity (machine). As in Kademlia nodeIDs are not certified, many nodes can be initiated on a single machine and afterwards these can be used in different attacks like eclipse and distributed denial of service attack (DDoS).

The technique used in LIKIR does not completely remove Sybil attack but in LIKIR user binds his identity to a particular nodeID by sending request to the CS. The user can have multiple identities as well and he can bind all these identities to particular nodeIDs and can run all those nodes on a same machine. This technique introduced a security measure which is the binding of user identity and nodeID with the help of certification service. It can help to restrict other attacks like DDoS and eclipse. LIKIR suggests that if human interaction can be introduced than it would be difficult for anyone to create many instances in an automatic fashion. It also suggests using OpenId verification methods to reduce this problem.

Storage attacks:(Index poisoning and content pollution)

Storage attacks can be index poisoning or content pollution. In Kademlia, nodes are responsible for different keys and store all the ¡K, V¿ pairs and return them when the values are requested. Value V can be a content, reference to other sources or it can be a set of meta-data. Index poising attacks can be done by adding massive number of bogus records into the index nodes. Attacker can remove the legitimate content with the corrupted content, e.g. a content not related to the key, reference to the wrong location.

In LIKIR this problem is removed with the help of cred attribute which is the binding of UserId to the key for which the content was inserted and to the hash code of the content. It is basically used to prove that a particular user is owner of this content. So in LIKIR whenever any node inserts content into the network they have to prove their identity by showing cred and same in the case of retrieving. In this way, the user can be tracked in case of corrupted data insertion on the network.

DDoS:

A distributed denial of service attack consists of a number of overlay nodes that generate huge amount of traffic towards a target node to make it unavailable or to other legitimate nodes. This attack is very common at present and can be achieved using the redirect technique [40]. In LIKIR this attack can be avoided with the help of cred attribute. So, in this way the owner can be traced and can be penalized accordingly.

Man in the middle:(MITM)

MITM attack is also very common in peer to peer networks at present. In this attack, the attacker intercepts the message and alters it according to his own needs. There has been research [41] that shows that at least half of the network has tendency to get attacked by MITM attack. To avoid this kind of attack there should be a check on both message integrity and credential authentication of the sender. In order to communicate with each other nodes has to create a session and the session consists of four way exchange of particular messages between two nodes.

LIKIR solve this problem with the help different parameters that are included in a session creation process. The parameters are singed AuthId, Auth, nonces between the nodes also AuthIds. The AuthIds are address specific because they contain the nodeID of the receiver.

LIKIR makes it very hard for an attacker to get the messages, alter them and replay them again to a certain node.

2.3.2 Peer-to-peer version control systems

In this section, our discussion will be based on a couple of papers [4], [5] that have been written on p2p version control systems. These papers give us a brief knowledge about the requirements needed to build p2p version control systems and how a p2p version control system can be designed and implemented. Later, this study will help us to design the functional specifications of a p2p version control system on top of LIKIR. They also proposed a prototype system each which gives us a very basic design specifications in order to build a large scale p2p version control system.

2.3.2.1 Distributed Hash Table Based Peer-to-Peer Version Control System for Collaboration [4]

This paper gives us design specifications of a p2p version control system and shows how a p2p version control system can be designed and implemented. The authors have also developed a prototype for the evaluation of functional specifications of the proposed system. The said prototype was developed with a minimum basic functionality and leaving rest of the functionality for the future development of the software and design.

The system has been developed using the layer based technique. In design, we will discuss different parameters that are necessary to understand the development of this system. These parameters include, design goals and different layers that are p2p layer, storage layer, meta-data, version management, work space and architecture of the system. We will discuss them briefly so that we can understand all segments of the system and the way system works.

Design goals:

There were three kinds of main goals that were necessary to keep into consideration in order to build distributed content distribution system on top of decentralized network.

First goal states that the system should support concurrent editing, read-modify-write style operations on objects that are used by more than one user. Also when the shared content is modified by different users and saved, then there can be collisions. So in this case there should be a mechanism to automatically or manually merge these changes.

Secondly, the system should support merits of p2p architecture that includes locality transparency and mobile user support. Thirdly, there should be a simple geographical user interface providing basic functionalities like, lookup(searching functionality), put(saving content), get(retrieving content) and versioning of the content.

Peer-to-peer Layer:

As we have discussed, Distributed hash table based peer-to-peer version control system for Collaboration is built using layer based technology.P2p is the bottom layer and provides all the services that any p2p protocol provides such as Message routing, peer lookup and atomic data manipulation. The p2p protocol which is used in order to build the prototype is chord [14]. Instead of Chord, other structured p2p message routing protocol can also be used such as Pastry [15].

Storage Layer:

The basic functionality of the storage layer is to store chunks of data into overlay network. This layer is situated on top of p2p layer so that the data coming from the upper layers gets stored into the distributed hash table along with the keys through which they can be retrieved again when needed. In this prototype data can be stored and retrieved using low level functions “put” and “get” respectively.

Metadata:

Metadata is used to describe the resource and is usually stored at the top of the head of the objects. In this case, metadata is used to describe the object that has been stored to storage layer and versioning layer. In storage layer, metadata contains data such as length, creator, creation time, modify time and etc. In case of versioning layer, it is used to describe the version number, owner, tag, branch, annotation etc. The Dublin core [42] metadata has been used to describe the objects in this prototype. The attribute set of the Dublin core was not enough so more attributes have been added to make it working to the solution.

Version Management:

Version management describes the techniques which are used for version management. There are basically two kind of approaches that are being used in industry and are,

- State based merging [43]
- Operation based merging [44]

State based merging approach has been widely used in past for many famous source based version management system like CVS [19], source safe [22] etc. This approach is only suitable for source based files and is not suitable for binary files.

Instead of state based merging, operation based merging is useful for all kind files because it ignores the actual data structure, provides better conflict detection and conflict resolution. In this approach, operations that have been performed over a particular file are maintained and then are committed. In this paper, operation based merging approach has been used due to its advantages of over state based merging.

Workspace:

Workspace is a storage space which is provided to the users to work on a task. In this case, the workspace is a collection of shared objects those are being edited locally by each user. When the shared objects are being manipulated by more than one user at different places then the changes can conflict with one another. To cater this, operation based merging is used by which conflicts can be resolved systematically.

Architecture:

As already discussed, the architecture of this p2p version control system consists of different layers and are shown in the figure 2.6. At the core we have p2p layer that help upper layers to execute low level functions of the p2p layer that include node lookup, put, get etc. On top of p2p layer we have storage layer and it includes storage service, metadata and versioning service. This layer helps to store the data on p2p network. Then we have the naming service which is situated on top of p2p services. Naming service can be directly accessed by all the layers.

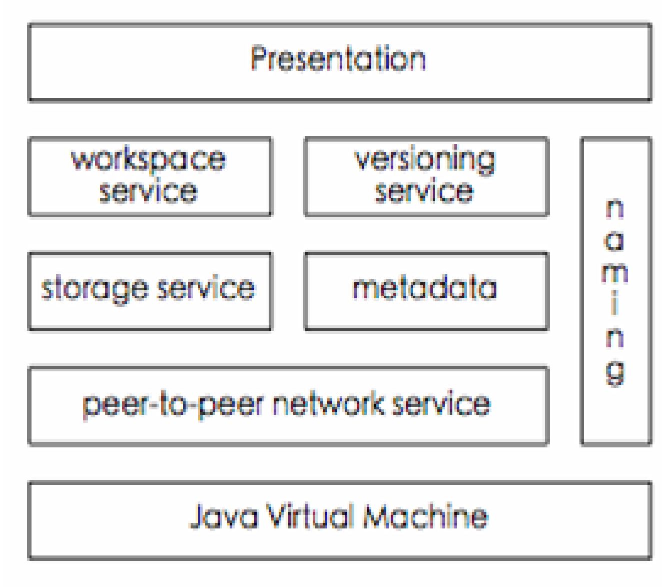


FIGURE 2.6: architecture p2p version controls system (adapted from [4])

2.3.2.2 Peer-to-peer based version control [5]

Like the previous paper, this paper presents another approach of purely decentralized p2p version control system. This paper also gives us all the required and basic functional details that are necessary for the development of p2p version control systems. The authors have also developed a prototype that has all the basic functionality that are required at the start and is discussed in this paper. This prototype has been built on top of FreePastry[43]. As described by the authors this solution is not limited to FreePastry other p2p frameworks who use DHT can also be used. We will briefly discuss all the details that are covered in this section.

Approach Used: Here, we will discuss the approach used to design and built p2p version control system.

Repository Distribution:

In this approach, the repository is distributed among the peers so each file and its revisions forms a repository item and each peer is part of repository. All the revisions are stored in an ascending order along with a metadata file that includes all the necessary information that is required to understand the file.

Fail-safe persistence:

As the peers are fully autonomous in p2p systems that makes robustness another issue because peers can leave and join the system any time they want. In DHT, data is not only stored at local peer but it is also stored at the neighbor nodes as well.

So in this system when any peer that has a copy of the content and the peer leaves or fails on the network then the neighbor of that peer is assigned as a replica peer in order to make the number copies stable on the network. This new peer is assigned the necessary files that should be available.

Check Out:

In check out, the content is retrieved from the network and this is one of the main functions of a version control system. In this approach if anyone wants to retrieve the latest artifact, he has to contact to the primary owner and then the primary owner sends the requested version.

Commit:

Commit is also one of the main function of any version control system. In this prototype the version control system is implemented in a way that when any changes are submitted by the user then these are routed to the primary owner of the content. Primary owner checks the validity of the changes that they are done on the latest version or not. If the changes are done on the latest versions then a new revision is created. Otherwise commit request is declined and the new revision is sent back to the peer forcing him to make changes in the latest version.

In case of concurrent changes, commit request is sent to the primary owner and then the request which receives first is treated accordingly and new revision is made. The request that comes afterwards is sent back because the version has already been changed and the changes have been done on to the previous version.

Chapter 3

Proposing new functionalities and Implementation

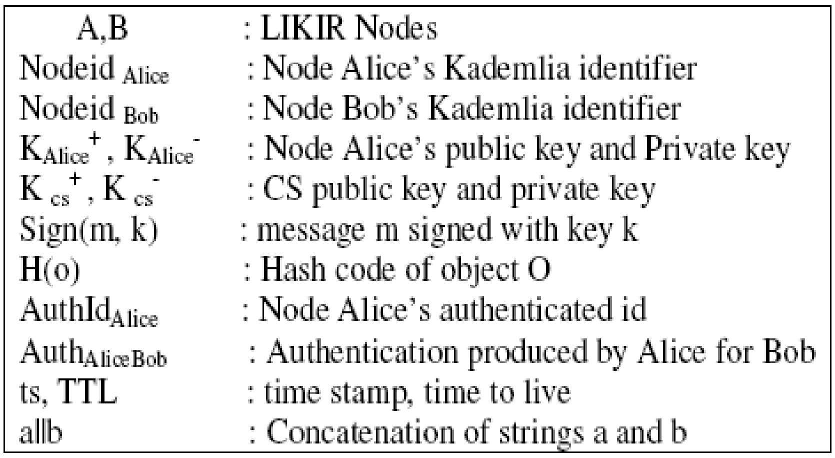
We have already discussed Kademlia in the previous chapter and we also have seen that what kind of attacks we have in Kademlia. In this chapter we will discuss the LIKIR framework which is built on top of Kademlia to remove most of the attacks and increase the performance of the protocol.

This chapter mainly deals with the analysis of LIKIR, LiCha and proposing new functionalities into LIKIR and LiCha. A part of the chapter also deals with the implementation of a new module in LiCha. LIKIR basically is a Kademlia like infrastructure and it improves Kademlia by providing security. Previously Kademlia was very prone to several of attacks which make Kademlia more difficult to be used for many professional applications. LIKIR remove almost all of the attacks by introducing the certification service, which certifies the users and adding additional attributes in messages to increase security. Currently, LiCha provides only some of the basic functions of a chat program like one-to-one chat, view off-line and online contacts, search and add contacts etc. In this chapter, we will describe LiCha and the proposed modifications for LiCha. Also the implementation of proposed combined chat module for LiCha.

3.1 LIKIR Analysis

3.1.1 LIKIR

LIKIR is a Layer identity-based Kademlia-like infrastructure. It is built on top of Kademlia and it provides a secure communication protocol, and an identity based scheme. It helps the P2P based applications to communicate securely and it also minimizes the possibility of many attacks on P2P based applications which are discussed in the previous chapter. Our emphasis will be upon secure communication protocol. IBS will be discussed briefly because it is not part of thesis work. The following notations are used throughout.



A,B	: LIKIR Nodes
Nodeid $_{Alice}$	: Node Alice's Kademlia identifier
Nodeid $_{Bob}$	: Node Bob's Kademlia identifier
K_{Alice}^{+}, K_{Alice}^{-}	: Node Alice's public key and Private key
K_{cs}^{+}, K_{cs}^{-}	: CS public key and private key
Sign(m, k)	: message m signed with key k
H(o)	: Hash code of object O
AuthId$_{Alice}$	: Node Alice's authenticated id
Auth$_{AliceBob}$	: Authentication produced by Alice for Bob
ts, TTL	: time stamp, time to live
a||b	: Concatenation of strings a and b

Figure 3.1: Notations used (adapted from [2]))

LIKIR enhances the join procedure, the node interaction protocol and content storage procedure defined by Kademlia [2]. In the coming sections, we will see the complete details of these changes and also the rest of functional specifications that are part of the LIKIR.

3.1.1.1 Initialization

Initialization is a process of starting an instance of a node. When a node wants to communicate with other nodes then at first it has to obtain a node id. During this procedure user also gets the bootstrap list (list of available users) that will be used later on to enter into the network. To get a node id peers have to send a request to the certification service (CS) along with user id and its public key:

$$NodeidReq = Userid_{Alice}\ K^{+}_{Alice} \tag{3.1}$$

User ID is the identity through which the user is presented to the other users on the network. User ID's validity can be verifiable through the CS. User identity may not be the existing identity in that case the CS maintains the user records and also verify them when needed. Otherwise when a user has an existing identity (Open ID URL or email Address), it can be verified through the CS to the respective service provider.

After receiving the request CS binds the user identity with the his public key and nodeID and then produce the following token,

$$AuthId_{Alice} = Sign(nodeID \| UserId_{Alice} \| K^{+}_{Alice} \| exp_{Alice}, K^{-}_{CS}) \tag{3.2}$$

This token contains the nodeID and it is randomly generated by the CS and expA is a timestamp that represents when the nodeID will expire. In response to a nodeID request from a user who is already registered will get the same AuthId that it received earlier because CS keeps track between the association of UserId and the AuthId. In this case the CS sends to the client the following response,

$$CS \rightarrow Alice : AuthId_{Alice}, Sign(bootstrapList, K^{-}_{CS}) \tag{3.3}$$

3.1.1.2 Join

After the initialization of the node, the next step for node is to join the network. For that Join operation is executed, which is the same as in original Kademlia. In join operation, the node executes FIND_Node RPC to find the nodes, it is also called node lookup. In this process nodeID is used in order to look up for a specific node or list of nodes; if the node ID is found, then the recipient returns the (IP address, UDP port, node id) of K nodes it knows closest to the nodeID. The nodeID's are selected from the bootstrap list provided by the CS during the initialization process. After the initialization, the CS is never contacted again, unless the node ID expires or the entire known users are off-line. So the node manages their own list of users, nodes can save the bootstrap list on to the network and can use again when it comes online again.

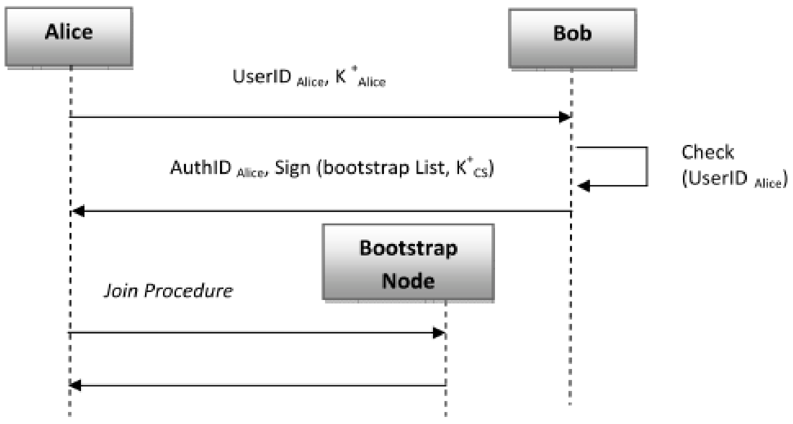


FIGURE 3.2: Node Initialization and Join Procedure

3.1.1.3 Interaction of Nodes

After initializing and joining the network when any node for instance Alice wants Bob to execute an RPC then both nodes have to use a proper process in order to execute the RPC successfully as shown in the figure 3.3. For this kind of communication Kademlia has a proper protocol communicate RPC to a specific node.

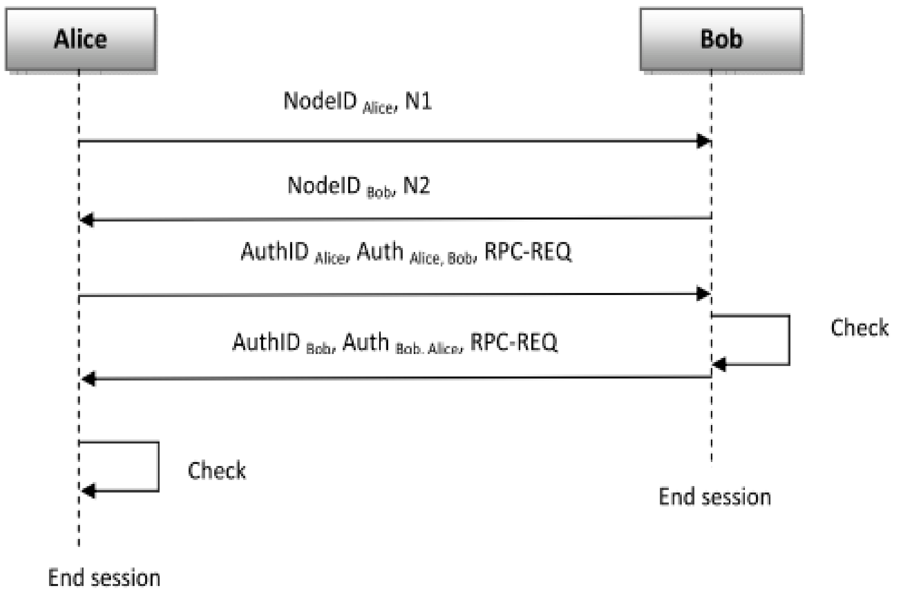

Alice
Bob
NodeID Alice, N1
NodeID Bob, N2
AuthID Alice, Auth Alice, Bob, RPC-REQ
Check
AuthID Bob, Auth Bob, Alice, RPC-REQ
Check
End session
End session

FIGURE 3.3: Four way interaction process

Once the client obtained the AuthId after that user do not interact with the CS unless AuthId expires or closed to expiration. Message from the CS also includes the bootstrap list and bootstrap list is combination of three attributes ¡NodeID, IP address and port¿ (also called triple). Basically it is a list of nodes that are present in the network and are active. So the user can use these nodes to start communication into the network.

It uses the four way communication in order to execute the RPC properly. We call this communication a "session" between Alice and Bob and is shown in the figure 3.3. In the first step Alice sends its nodeID and N1 (Nonce) to Bob and on the other side when Bob receives the message it sends back its own nodeID and N2 (Nonce). N1 and N2 are randomly generated nonces which are basically used to protect from replay attacks. Now Alice will send the RPC request which she wants Bob to execute and is done using the message including AuthId Alice, Auth Alice, Bob, RPC-REQ and when the message is received on other side Bob checks the message whether the message is valid or not (means it has everything that is required in the message) if the message is valid then Bob executes the RPC and sends back the response AuthId Bob, Auth Bob, Alice, RPC-REQ.

Now the AuthId Alice has a special token produced by the CS and it includes,

$$AuthId_{Alice} = Sign(NodeID \| UserId_{Alice} \| K^{+}_{Alice} \| exp_{Alice}, K^{-}_{CS}) \tag{3.4}$$

The third and fourth message includes, the RPC request and the RPC response respectively from each other and messages include,

$$Auth_{Alice,Bob} = Sign(NodeId_{Bob} \| N2 \| H(RPC - REQ), K_{Alice}) \;\; and \tag{3.5}$$

$$Auth_{Bob,Alice} = Sign(NodeId_{Alice} \| N1 \| H(RPC - REQ), K_{Bob}) \;\; repectively \tag{3.6}$$

3.1.1.4 Content storage System

Storing content is very important in p2p systems. Because while storing the data you have to take care of many things for instance storing content to proper peer, content Security, protection against content forgery etc. LIKIR implementers also took care of this; every other RPC follow the Kademlia definitions except the store RPC. In Store RPC, if a node Alice wants to store content say obj into the DHT then that node has to find out k nodes closest to the content's key, that is described as follows,

$$Alice \rightarrow Bob : AuthId_{Alice} + Auth_{Alice,Bob} + StoreRPC \tag{3.7}$$

$$StoreRPC = K \| Obj \| Cred \tag{3.8}$$

$$Cred = Sign(UserId_{Alice} \| K \| H(Obj) \| ts \| TTL, K^{-}_{Alice}) \tag{3.9}$$

As we already discussed most of the parameters that are included in the message except the store RPC which includes K, content to store and the Cred. Cred binds the UserId to the key for which the content was inserted and to the hash code of the content so that it would be possible to prove that the owner inserted the Obj having key k. A node performing lookups for contents related to key k, receives all the nodes that are responsible for the content related to key k.

LIKIR also maintains a list of users which are not behaving accordingly and sending malicious or polluted data. These users can be identified from the information available in Cred attribute and then afterwards penalized. This simply can be done by announcing the nodes and instructing them to refuse all incoming requests from them and it can be identified from AuthId. This approach really reduces the propagation of polluted content.

3.1.1.5 Bootstrap List construction

Bootstrap list consist of nodes that are in the network and are still active. Certification service creates this list and then afterwards forwards this list to each new incoming node. Nodes then can enter into the network contacting at least one of node from the list. Actually bootstrap list is called triple and are stored in the server cache

$$CacheEntry = (NodeId, IPaddress, UDP\, port, ts) \tag{3.10}$$

The bootstrap node selection is a problem inherent to the fully distributed nature of the P2P networks [2]. This process should prevent the attacker to manipulate bootstrap information and let nodes to join another parallel malicious network. Kademlia does not face the node selection problem. But still LIKIR provides a more practical approach to cater this problem with the help of certification service.

As we earlier discussed that certification service maintains a list of active users and then CS probes all the nodes in the list one by one. CS does that with the help of a node that is solely for the CS with a self signed AuthId. CS implements least recent cache replacement policy otherwise the active nodes are never removed from the list. When the list is full then the least recently seen nodes are pinged. If they don't respond, then these nodes are removed with the new nodes. On the other hand if they respond then these can be moved to the tail and the new found nodes are discarded.

3.1.1.6 IBS (Identity based Signature)

IBS is actually a cryptographic technique which is used to generate a key pair and the public key which can be easily found with the help of ASCI string. This technique can be used to get rid of traditional Public key cryptography (RSA:Rivest, Shamir and Adleman). It's a four step procedure which starts by setting up a pair of master keys MK+ and MK- then private key is generated. Then Alice can sign the message with her private key and on the other end signatures are verified.

3.2 LIKIR Problems

Like other DHT protocols LIKIR also faces some minor drawbacks are described as under,

- CS can be a single point of failure. But certification service is not used always, it's only used when either the AuthId is expired or close to get expire.

- PK management is expensive and introduces enormous overhead. Everyone knows that it's expensive and resource consuming but still it has to be used to provide security and is largely accepted.
- IBS also Require PKG which makes it single point of failure due to key escrow problem.
- IBS looks perfect in theory, not in practical environment.
- It introduces extra overhead and it's expensive.

3.3 LIKIR modification plan

Previously, we have seen all the required information regarding the LIKIR and the Kademlia to propose the specified module. Now, we will concentrate solely to solve the problem as discussed in the start of this report as part of proposing suggestions in LIKIR. Specifically, we have to propose an architecture for user information module that can be added to LIKIR protocol without modifying anything in LIKIR.

For that purpose, an architectural scheme is introduced which keeps in view all the requirements that are necessary to implement this module.

3.3.1 Description of user information module (UIM):

The User Information Module stores information about the users and the information can be personal or specific as required by the application. For instance, the information can be a node id, name, country, date of birth, sports, activities etc. For some application, storing user information is necessary because it can be used to acquire information about any specific user or other users of this application. This information can be used to provide better services to users according to their profile or interests. User's identity is at the heart of LIKIR design, so by providing a dedicated module to manage user information will be very useful. Later on, this data can be used to identify user's interests, hobbies etc. This information can also be used to built new applications on the basis of user interests. The UIM, incorporates a set of classes and these classes work separately from LIKIR protocol, i.e., their implementation will not interfere with the current LIKIR functionality.

The UIM can be used with any application easily without having any problem. Also it is not obligatory for the applications to use this module. In fact, its usage is application-dependent and UIM works like an application add-on.

The UIM is shown in figure 3.4 and basically consists of an interface that allows communication between the applications (like LiCha) and LIKIR. The user information module also consists of a class that will implement a data structure for using different functionalities (storing, retrieving etc.). The Get and Put functions among other functions of LIKIR are used to get the latest version of the content and to store the content on the network.

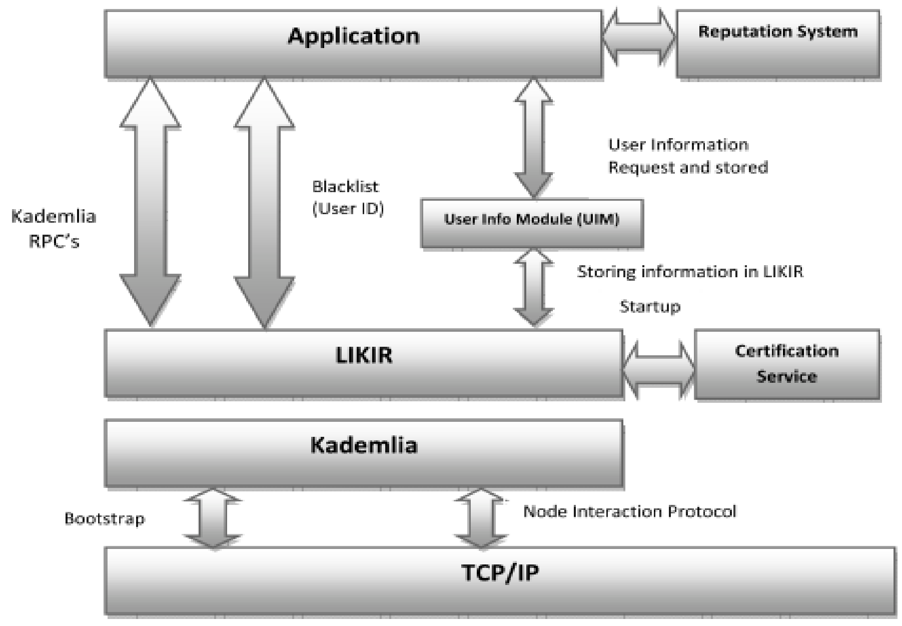


Figure 3.4: User Information Module

3.3.2 Getting User Info

The process of gathering and storing user information can be initiated at any time during the life cycle of the application. To generalize this solution, there is no restriction onto the applications to provide the information immediately after the node initialization process.

In fact, this information can be gathered and saved at any time during the application life cycle. For instance, an important attribute that is stored is verified UserId. This can be collected immediately after the user has got the verified node ID from the Certification service and can be used to identify user's information.

3.3.3 Storing using information:

When user information is gathered then it needs to be stored so that any other interested user can get this information, even if the user who is storing the information is off-line. The gathered information will also be encrypted using the appropriate encryption technique, or the one which is already being used inside the LIKIR protocol to encrypt options, contact and buddy list. Later on, this encrypted information is stored on to the LIKIR network in the same way the options, contact and buddy list are stored in LiCha but with different access parameters(who can access and what). At the application level, a copy of the user information is also stored in an XML file by the application in the local system. The rationale is to provide a facility to restore local user information at any time during the interaction. So it is a kind of backup facility for the users and can also be used when the application is started in the future.

3.3.4 Editing User Information

A user can edit her information whenever she wants to. For that, the UIM will implement some methods (in the implementation class of the interface) which will initially help users to retrieve, make some modifications, and then store the user information inside the network again. So, the other users and the applications can get the updated information. Also, the copy of the XML file will be replaced with the new one which is stored locally.

3.3.5 Privacy matter:

Privacy of the information is always very important in every application. Handling privacy at the protocol level can make the protocol more complex. UIM allows the designers to maintain privacy at the individual level.

Privacy of the information can be handled by only allowing certain information to be available to the users for specific reasons. For example a user wants his <name> and <surname> to be disclosed and used for searching. On the other hand user do not want to disclose his <country> and <birth_date> also to be used for search. User wants to be known only by his <id>, but no other information disclosed. Finally I want <religion> to be fully protected, which means that nobody can see it, but, for instance, it can be used for statistical purposes: to see the user percentage of different religions: 80%<void>5%<Christian> 10%<Buddhist> 5%<Muslim> etc.

All of this required the collection of additional information from users, and on the other hand, full control of the user over his stored information. He can decide which information should made available for others to see.

3.4 Proposed implementation of user information module

As mentioned earlier that UIM consists on a number of classes and these are described as under,

UIM.dataStructure This class will provide the data structure to hold user information and the number of fields in this class can vary from one application to another application. A set

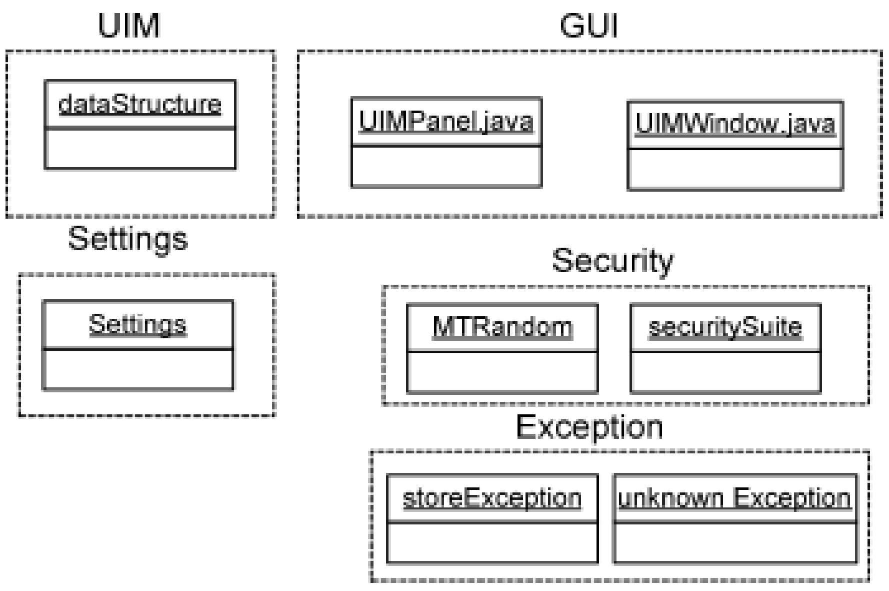


Figure 3.5: User Information Module Classes

of general parameters can be defined and latter on, if there is need to add more attributes then "UIM.dataStructure" class can be customized accordingly.

UIM.gui It will include the geographical user interfaces that are related to user information module. GUIs depend on the application because different applications have their own specific user requirement.

UIM.Exception This package includes a couple of classes that will handle the different exceptions and more specifically two kinds of exceptions one that deals with storing the data and the other is for rest of the exception that can occur.

UIM.security It will provide a random number generator (the Mersenne-Twister[46]) and a Security Suite class, that latter keeps the user DES key, generated by the password provided during login and will expose methods to encrypt and decipher the content.

It will use some classes from the application (LiCha) itself which are message; security classes and the rest of the classes are the specific user information module classes. We can define separately all those classes and we can also use the already created classes with certain application on which UIM module is going to work.

3.5 Usage of user information module in real life

User information can be used for many things; it can be used to provide better services to the customers. For instance, a user is using media services from a content distribution company then on the basis of user profile company can send promotions and content related to users interests.

3.5.1 New applications

On the basis of users interest new applications can be created and these kinds of applications can be games or other applications and further more these kinds of applications can be used in social networks. We can take examples of different social communities that are developing new applications on the basis of user interests like facebook, orkut, hi5 etc.

3.5.2 Scientific Research

User information can be used to conduct specific scientific research and this research can be anything related to the user information like user behaviors, hobbies etc. with help to this information, service providers can provide better services and also this information can be used to identify the behavior of users that are using the specific service.

3.5.3 effective marketing

On the basis of user interests user can get various updates and offer from different companies that will help user to get the information easily and quickly. The offers can be sent to the user via email, cell phones or can be shown somewhere in the GUI.

3.5.4 sharing data with third parties

User information can be shared with third parties in order to make money. Most of the companies sell the user information to the third parties to gain money. Further these companies use provided information to conduct surveys, send advertisements by email etc.

3.6 Analysis of LiCha

LiCha is chat client which has been built on top of LIKIR. It's the first application that will try to mark this protocol in the P2P industry. It's still in development phase but its basic version is available and is free to use. The basic version gives some basic functionalities like one-to-one chat functionality, management of the contacts, contact status info, searching and adding of client and user customized changes. There are many other functionalities that are under consideration for future version of LiCha. LiCha can also be used as a starting point to understand the requirement necessary for an application to interact with the LIKIR. Here in this section, we will mainly analyze the LiCha. In general, we will also see the requirements of an application to interact with LIKIR framework in general.

3.6.1 LiCha

Every LiCha client relies on a LIKIR node. Every LIKIR node is identified with her UserId on to the network. In case of LiCha, LIKIR node is used as an account identifier for a chat client. LiCha uses commands to insert three types of content into the DHT that includes the user options, the client contact, and the buddylist. User options contain the local client customization preferences; for now they include only the background color of the GUI. The client contact trivially maintains the TCP socket address of the chat service, a status specification (offline/online) and a port on which a client listens for incoming applicative ping messages. The buddy list is a list of UserIds of known friends. During this chapter, we will refer options, contact and buddy list to a generic user Alice respectively with optionsAlice, contactAlice and buddylistAlice. Like ordinary chat clients, LiCha boot phase starts by displaying a simple user interface which allows a generic user Alice to insert her UserIdAlice and a password. When the required data is inserted, a LIKIR node is instantiated and LIKIR node uses the normal procedure as described earlier. If it is a first login, the CS is contacted to retrieve a signed AuthId, and afterwards the LIKIR node is bootstrapped. Conversely, if the user logged in previously as well then a file containing its AuthId (placed in the "Nodes" directory) is loaded from the file system and the CS is not queried. When the underlying node has bootstrapped and the AuthIdAlice is gained (by loading it from local storage, or, in case of "first boot" scenario, by contacting the CS), the application calls LIKIR layer APIs to search the optionsA on the DHT (step 1). The LIKIR GET method is invoked, specifying "LiCha" as application-identifier string and "true" in recent parameter, to get only the latest version:

$$Get(Options_{Alice}Key, \text{``}LiCha\text{''}, UserId_{Alice}, true) \tag{3.11}$$

$$Options_{Alice}Key = H(UserId_{Alice} \| \text{``}optionsSuffix\text{''}) \tag{3.12}$$

The index side filtering feature only allow the application to ask only for contents published by its own node. This possibility reduces the size of a set of returned contents and assures that the information retrieved is completely safe, because signed credentials grant its un-alterability. The recent (which is true in this case) parameter allows to get only the last options stored, even if many different versions are kept in the DHT. Furthermore, LiCha encrypts user options with a symmetric key extracted from the password inserted by the user. Options encryption grants privacy of data, preventing malicious users to access reserved user info. Once, the user options are retrieved. The application tries to decrypt options using its symmetric key and if the result is an object encoded in a correct format, LiCha loads the option otherwise it shows an error message and quits. The encrypted buddylistAlice is retrieved in the same way.

$$Get(buddyList_{Alice}Key, \text{“}LiCha\text{”}, UserId_{Alice}, true) \tag{3.13}$$

$$buddyList_{Alice}Key = H(UserId_{Alice} \| \text{“}friendSuffix\text{”}) \tag{3.14}$$

Once options and buddylist has been retrieved, LiCha builds a proper contact by storing its socket address, its ping port and its status specification. LiCha inserts this information into the DHT (step 2) by doing so the client announces to the network that it is online. The LIKIR PUT primitive is called, specifying a default settings-dependent time to live.

$$Put(Contact_{Alice}Key, contact_{Alice}, \text{“}LiCha\text{”}, defaultTTL) \tag{3.15}$$

$$Contact_{Alice}Key = H(UserId_{Alice} \| \text{“}infoSuffix\text{”}) \tag{3.16}$$

The next boot phase consists of retrieving the contacts of all the friends filled in the buddy list (step 3). The reader must notice that, for a generic friend F, many contact published at different times could be retrieved. But, again, the filtering features allow to easily detect the last valid contact published. The GET method called is parameterized similarly as it was in step 1:

$$Get(Options_{F}Key, \text{“}LiCha\text{”}, UserId_{F}, true) \tag{3.17}$$

Finally, LiCha sends an applicative ping message to each online contact collected (step 4); since a peer can unexpectedly fail or go off-line after inserting its online contact. Pings verify that probed clients are really up, but also allows informing them that the querier node has gone online. Once completed the four steps described, the LiCha client has all the information to create the lists of online and off-line friends and it is ready to serve user requests. LiCha clients who know each other's contacts can simply start chat sessions without involving the LIKIR layer. A ping message is regularly sent to all live contacts to check their activity, because any peer can always fail silently. As we stated before, when an off-line LiCha client goes online, it pings all its friends, thereby updating their online and off-line friends list. LIKIR support is used to search for any new friend X; if contactX is found on the DHT then X can be added to the LiCha buddy list. The user contact is periodically refreshed on the DHT accordingly to the defaultTTL specified, in order to prevent old contact expiration. Finally, when LiCha is terminated by the user, the encrypted user options and buddylist are put in the DHT by calling the LIKIR interface methods as shown before. At the next ping round, the friends of the departed node will check its off-line status and update their buddy list.

3.6.2 Chat procedure

To start a chat with another client user has to select the desired client and then press start chat button. The chat will be started normally as it is started in other chat services. On the backend, the starter of chat sends an invitation message to the selected client and then the client who receives the invitation message will make some changes in the message by putting his own address in the server field and changing the opcode value. Afterwards the message is sent back to the client who initiated the chat as a confirmation that the client is ready for chat and then on both sides chat room object is created which contains the ip addresses, ports and other related information of clients participating in chat and is as shown in the figure 3.6. Chat room object is used afterwards for the whole communication process.

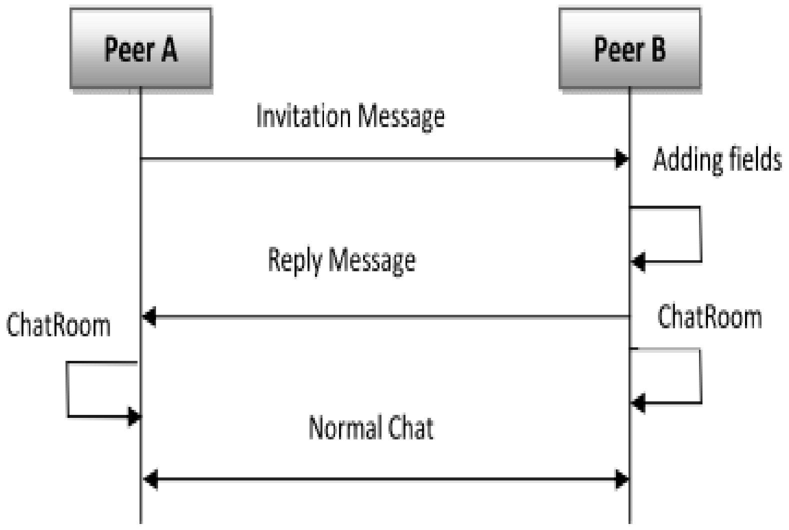


Figure 3.6: Simple LiCha Chat

3.6.3 LiCha package description

LiCha packages are basically structured according to the Model View Controller pattern.

unito.licha.gui: This package includes, the LiCha graphic interface specification. LiCha GUI has three windows: the login window (to insert username and password), the main window (that shows the buddy list and the menu to find new friends) and the chat windows (to interact with other users). For every GUI element 2 classes are given: the window class and the panel class.

The window is a JFrame, while the panel is a JPanel instance, which contains all the graphic elements.

unito.licha.controllers: There is a controller for every LiCha window. The LoginController is simply an ActionListener of the LoginPanel class, and is used to handle the user login request. The CoreController and the ChatRoomController are "bridge" classes between the model and the view. The CoreController is an ActionListener of the MainPanel GUI class; when a button in the main window is pressed, the CoreController instance calls the proper method of the ChatCore class, which is the heart of the LiCha application logic. CoreController is also an Observer of the BuddyList class instance maintained by the ChatCore. When the buddylist changes its status (for example, a new friend is added to the list), the CoreController is notified and the GUI is changed accordingly (e.g. the new friend name is displayed in the window). The same is for ChatRoomController, which are an ActionListener of ChatPanel class and an observer of the ChatRoom application logic class.

unito.licha.core, IO, messages: This package implements the application. The ChatRoom class is the hub of all LiCha application logic. It keeps references to the two main data structures (the BuddyList and the user Options) and to input/output services. When the

ChatRoom is instantiated and the start() method is correctly executed, LiCha is online: the underlying LIKIR node is running and the application input/output services are active. The BuddyList and Options classes are very simple data-wrapper classes. They can be extended or enhanced simply. The BuddyList is a list of FriendContact instances; every FriendContact contains all the useful data to contact the friend LiCha clients. I/O classes are a bit more complex. Every LiCha client has always two daemons running: a ChatServer and a PingServer, listening on two randomly chosen TCP ports. The ChatServer must receive all the incoming ChatInitMessage requests, send back a ChatInitMessage response and properly instantiates a ChatRoom object, then the communications during the instant messaging session are completely delegated to the ChatRoom, that allows to send a new text message (sendMessage(String msg) method) and to abort the session (exit() method); obviously, the ChatRoom listens for incoming text messages sent by the peer client. A ChatInitMessage can be sent just calling the ChatCore startChat(String UserId) method. This method searches in the buddy list for the FriendContact corresponding to UserId, then sends a request, waits for the response and finally instantiates a ChatRoom object. The PingServer listens to all incoming PingMessage requests and replies with correct responses. When a ping is received, the ChatCore BuddyList is properly updated. Symmetrically, from client side, we have the PingClient class, that send a ping, waits for a pong and updates the buddy list accordingly to the response (if no response is received the contact is set as "off-line"). When the ping/pong exchange is over, the PingClient instances dies. A PingClient can be instantiated just calling the ChatCore ping(String UserId) method. The last I/O manager is the Refresher class, whose run() method is executed periodically by a Java scheduledExecutor. The Refresher start() method starts this periodic service. The Refresher executes only two activities: first it pings all the known alive contacts to check their activity, then inserts in the DHT a new local contact. When the chat is closed (ChatCore exit() method) all the steps described in the "LiCha general description" section are executed

unito.licha.settings: This package contains two simple classes to retrieve settings from the config.properties file. The PropFinder class provides an interface to access to properties just giving the public strings contained in the Settings class as parameter of the get(String name) method.

unito.licha.util: This package includes only the ContentManager class, that provides static methods for Object serialization and deserialization.

unito.licha.security: This package provides a random number generator (the Mersenne-Twister) and a SecuritySuite class. The latter keeps the user DES key, generated by the password provided during login and exposes methods to encrypt and decipher the buddylist and the options.

unito.licha.exceptions: This package includes an exception thrown when an unknown contact is searched in the buddylist (UnknownContactException) and an exception that models an error during the LiCha initialization phase (i.e. during the ChatCore start() method)

unito.licha.LikirChat: This package provides the main method to instantiate a login window and its controller.

3.6.4 LiCha improvements proposed

There are some proposed enhancements which are a mixture of some general chat functionalities and some specific functionality related to the P2P LIKIR.

General functionalities:

1. **Geographical user interface(GUI)**:
    - User Login should be like other chatting services like MSN, yahoo etc.
    - Contacts can be in the same List. Online and off-line contacts can be identified with different Icons.
    - Providing option to manage contacts into different categories. (Family, friends etc)
    - User Status Options (Like busy, idle, away etc.)
    - User personal customized message option ()
    - Customization of font style

- Display Picture (optional)
- Smiley (optional)

2. **Specific functionalities:**

   - Audio chat
   - Video Chat
   - Combined chat(chatting with 2 or more persons in the same Widow)
   - See user detail (IP address, Node ID, User ID etc.)
   - File transfer
   - File sharing (To some specific users and all the user's)
   - Enhanced user searching criteria (With some other fields as well like IP Address etc.)
   - Contact list can also be stored in LIKIR network

3.6.5 implementation of proposed solution

This section includes the description and implementation of the module which was selected and afterwards built after mutual discussion between the stakeholders (University of Trento and university of torino). This module provides the functionality of the combined chat in the previously built chat application LiCha.

3.6.5.1 Development goals

The development goals for this combined chat module are as follows,

- Contacts selection
- Sending invitation message
- Confirmation message
- Connection Status
- Message transmission
- Off-line notification

Contact selection:

In contact selection clients are capable of selecting other online contacts for combined chat. The selected clients are than used for starting the combined chat.

Sending invitation message:

After the selection of contacts invitation messages are sent to all the selected contacts. All the contacts are always listening for the chat requests so they receives the chat request and than sends a confirmation message back.

Confirmation message:

When invitation messages are received the contacts checks the message validity. If the message is valid then the invitation message is modified according the given standard. The modified message is sent back to the sender so that the chat can be started otherwise it will throw an error message.

Connection status:

When the sender receives response of the invitation message, than the all the participants open their combined chat windows automatically including all the contacts who were selected in the start for a purpose of combined chat. If you see all the contacts that were selected earlier, it means you are connected to all of them and now can send message to them.

Message transmission:

After checking the connection status all the participants can send messages to each other. There can be no error during the message transmission as we already had checked the connection status among the participants. There can be error during the transmission of message but it can be only due to network or any other problem after the initiation of combined chat.

Off-line notification:

During the combined chat process any contact can go off-line. The message sender can get error messages, if the off-line contact is not updated in the combined chat. So whenever any contact get off-line, it is removed from the combined chat and a message is sent to all the participants.

3.6.5.2 Development approach

This section deals with the development of proposed solution i.e., a combined chat module. With help of this module users can have combined chat with other users in their contact list. More explanation of this module is available in coming sections.

For the development of combined chat we have used incremental approach because with this approach developer can learn and develop at the same time. In the incremental approach, the developer starts with a smaller subset of the requirement and then iteratively moves on to the bigger subsets of the requirement until the full requirements are not matched.

3.6.5.3 Development environment

We have developed the module using java because previously LiCha was built using java due to its obvious advantages. The specific version of the java that has been used was version 6 update 17 which is normally being used these days. LIKIR frame work was also used to access different functionalities of the LIKIR frame work.

3.6.5.4 Module design

Combined chat module is not so different with the normal chat in the LiCha. Combined chat module can be said, the modified version of the LiCha chat. Let say, we have three users A, B and C as shown in the figure 3.7. For instance Peer A wants to start chat with peer B and C. To start combined chat peer A will send invitation messages to B and C one by one. When peer B will receives the message it will perform a number of operations that includes filtering the contacts received in invitation message. It will remove unnecessary contacts and after that, it will send the invitation message to filtered contacts. Before sending message, it will check on the basis of their names that which name had bigger ASCII value than other one. It will check if the contact is bigger than its own ID then it will send the invitation message to peer C and if not then it will create ChatRoom object for both peer A and C. On the other end peer C will create ChatRoom object for peer A and B. After creating all these ChatRoom object peers can chat with each other.

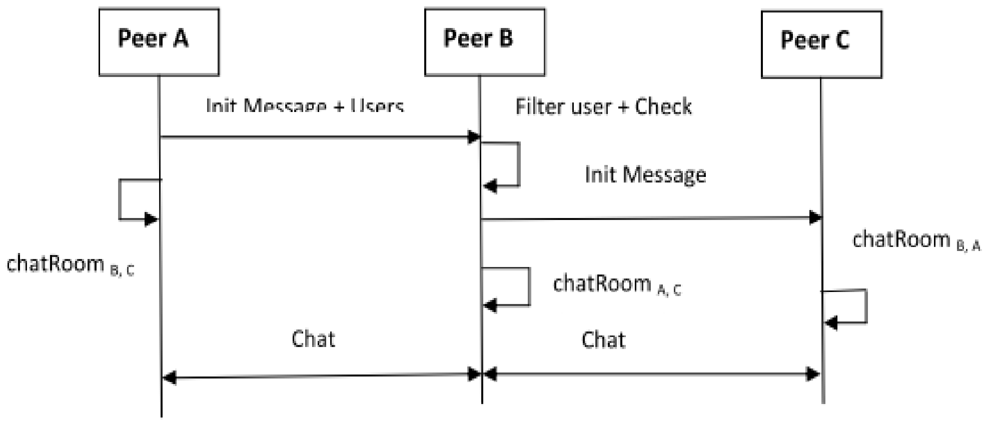


FIGURE 3.7: Module Design

3.6.5.5 Classes:

Combined chat module comprises of different classes that were used, other than its own classes. We also have made changes in existing classes to construct and make this module working. Every class has their own functionality and is explained as under,

CombinedChatRoomController: This class is a bridge class between the model and the view similar to the "ChatRoomController" class. The only difference between both classes is "ChatRoomController" is only for ordinary one-to-one chat and "CombinedChatRoomController" is for combined chat.

Combined Chat window: This class deals with two classes that are "combWindow" and "combPanel". These classes are the responsible for GUI of the combined chat.

Chat core: To make combined chat module working, there was need to make some additions in the chat core class. In this case these changes were the addition of "startCombinedChat" function which starts new chat sessions with a list of friends. Previously "startChat" function creates only single session with the selected friend.

Chat Server: In this class major additions took place that includes the identification of incoming combined chat request, filtering(on the basis of ASCII value of the name) the contacts to whom send invitation request and then create chat room objects related to each node and then pass these nodes to respective controller class to initiate the chat.

Core Controller: In this class, we added the ability to catch the action listener related to specific button. This is the where combined chat module is started.

3.6.6 Evaluation

This section deals with the evaluation of combined chat module based on the goals that have been discussed earlier. Evaluation is composed of certification service, DHT service, chat clients and evaluation environment. Certification service is used to provide authenticated nodeID and DHT service is used to start the LIKIR overlay network. Once the CS and DHT service are started, then we can start the chat clients. Before all staring all these, we have to select one of the environment. Then clients will start communication between each other in different environments and the number of chat clients are same in all environments. We have three kinds of environments and are described as under:

- Local host Evaluation
- Local area network (LAN) Evaluation
- Wide area network (WAN) Evaluation

Local host evaluation:

Local host evaluation has been done on a local machine with three peers(Peer A, Peer B and Peer C). Peer A starts the combined chat by selecting the contacts (Peer B and Peer C). So before Starting the combined chat peer A has to send invitation message to both the peers and wait for the response. When peer A gets a desirable response then the combined chat is started among the peers as shown in the figure 3.8. Peers now can send transmit chat messages to each other successfully.

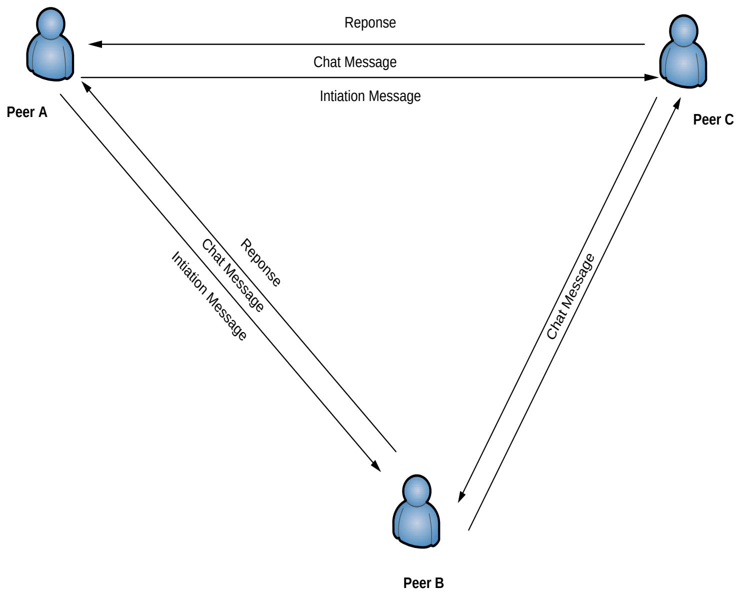


FIGURE 3.8: Local host evaluation

Evaluation of combined chat at local host one the basis of described goals was successful and the results can be see in the table 3.1.

LAN evaluation:

In LAN evaluation, peers are distributed on a local area network and the same number of peers are used in this scenario. A special LAN envoirnment was built to evalute the combined chat applicaiton. The difference between LAN and local host evaluation is obvious. One is being tested on local machine and the other one is tested on local are network and as shown in the figure 3.9.

Evaluation of combined chat at local are network one the basis of described goals was successful and the results can be see in the table 3.1.

WAN evaluation:

In this evaluation, peers are located on different places in the wide area network. Specifically, two peers A and b were situated in Sweden(Stockholm, Malmo) and third peer was situated in Pakistan as shown in the figure 3.10.

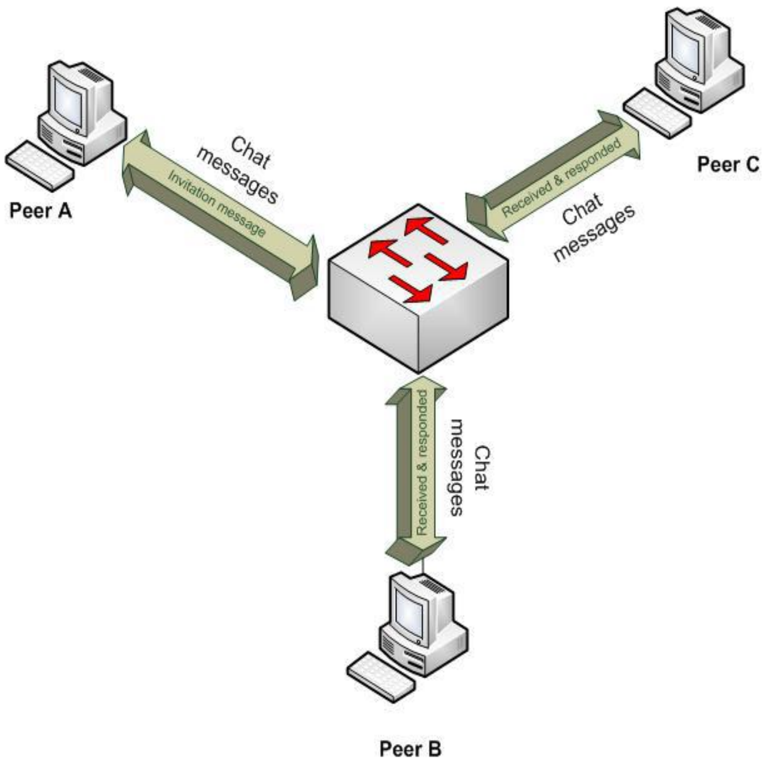


FIGURE 3.9: Local are network Evaluations

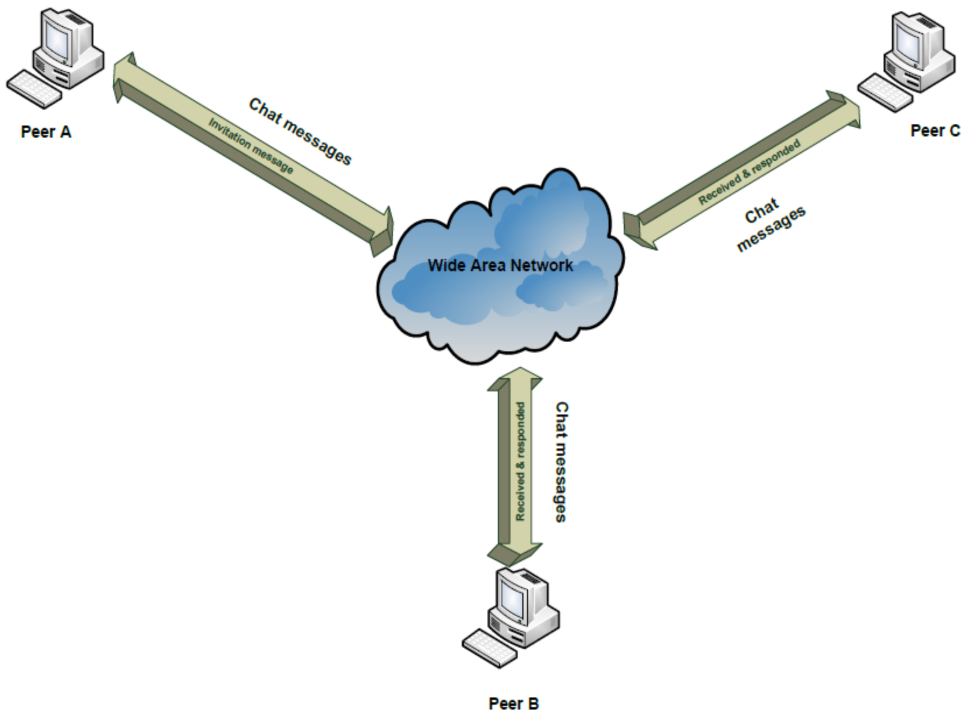


FIGURE 3.10: Wide area network evaluation

All other settings are the same except certification service which in this case is hosted on public IP. Then the evaluation begins by selecting the contacts and pressing the chat button to start chat. The result turned out to be the same as it was in all the other cases and is shown in figure 3.1.

Table 3.1: Evaluation Results

Properties	Local host	Local area network	Wide area network
contact selection	Yes	Yes	Yes
Sending invitation message	Yes	Yes	Yes
Confirmation message	Yes	Yes	Yes
Connection Status	Yes	Yes	Yes
Message transmission	Yes	Yes	Yes
Off-line notification	Yes	Yes	Yes

Chapter 4

Peer-to-peer version control system‘s architectural design

Version control systems have been a de-facto standard for the software development industry since many years. Many version control systems had been introduced in the last decade that includes Subversion, CVS, and RCS etc. Version control systems became a necessity, especially for the software development teams and other companies. It made software development more manageable, flexible and provided new approach to make software development easy even from the remote sites. It helped software industry to develop applications in an environment in which many developers work on a same project and are geographically dispersed. All these proposed systems were based on client-server architecture, which is a very good technique in many situations, but it’s not appropriate to provide scalability, availability, robustness etc. in many situations. So at present Peer-to- peer systems are taking over from client server architecture in many applications due to the advantages of the P2P architecture over client-server architecture.

There is a lot of research going on to P2P version control systems. Furthermore, a couple of P2P version control systems [4], [5] are available and discussed in detail in chapter 2, but unfortunately they are not mature enough to be used. There is still work going on at these projects to make them mature so they can be used in future or can be a source to make a professional P2P Version control System. We will also discuss these design alternative briefly in this chapter and see the pros and cons of each approach. Afterwards, we will see the differences between the alternative design approaches and the one which is being proposed in this chapter.

In this chapter, a new approach is being defined using the best features available at present. We have analyzed the already present centralized, distributed and P2P versions control systems and then proposed a P2P version control architecture based on best possible approaches available at present. This design is based on LIKIR protocol so it inherits both the advantages and disadvantages of LIKIR.

4.1 Design requirements of Peer-to-peer version Control system

4.1.1 Design Goals

The design goals of the P2P version control system are at first to provide all the basic features of version control like checkout, commit, concurrent commits and also to provide distributed environment (Workspace) on the basis of LIKIR. In future more functions can be added according to the need.

4.1.2 Check Out

Check out is a process in which you will copy a project or a file at your local machine in which you are interested to work. To get this file a peer has to search the particular file from the repository. When the particular project/file is searched then the peer will get a number of results about the peers hosting that project including the project owner. Normally, the project files are being copied from the owner because only the owner and his replicas have the current copies but in this case the project files can be downloaded from any peer shown in the search because all the peers have the recent copies. If a peer is looking for a specific version the request will be sent to the owner because the owner has the information about all the versions. So owner will return the specific version requested. After copying, the project files are stored locally.

The system will store all the information regarding users who checked out the same project or working on to the same project. It is because when there are some changes in the project then they can be propagated to all the users; so that peers can have the updated files to work on. That can make the probability of conflict occurrence very low.

When you will search for a project, the result will show you all the details including owner, version number, time, comments, participants, Peer name etc. that can help you to pick the version from a peer you like or the one you are working with.

4.1.3 Commit

After making all the changes when a developer/peer submits the project files to make a new revision. The request is sent to the immediate user with whom you are working with. Then afterwards it’s sent to everyone who is participating in that specific project. In the request, the peer will not send the edited file and the revision number of the project because we are not using the state based merging approach instead we are using operation based approach, which will be discussed later in detail. In this approach, we only send a file that contains the actual operations that were performed. To make things simple the file that contains operations is routed to the immediate user as discussed above. The user checks the file, if it is a valid commit request then a new revision is made and a success message is sent to sender and then the revision is also stored on different peers on the network. If not the specific message indicating the conflict will be sent back to the user if, and only it cannot be solved automatically.

User intervention cannot be ignored; it has to be used at some stage when the conflict cannot be resolved automatically.

4.1.4 Concurrent commits

In this situation there can be a couple of scenarios like only 2 peers who are working onto the same project/document or more than two users. In case of 2 peers both will send updates to each other about the operations performed to make revisions (even if both are not the owners), and when the owner comes online he will get all the updates they have. So if there is no conflict between the operations performed by the peer then new revision will be created without any conflict. If not then specific message regarding the conflict is sent to the peer who committed the project/file. So that he/she can manually edit the file according to the message.

In case of more than 2 peers as shown in the Figure 4.1 when user A and user C working together on a project but actually the project is also being shared by other users B,D,E and F. Both A and C are working together on the project. User A made some changes and then submits its changes to the user C. User C will then check to commit request and check, whether it’s not overlapping with its own operations. After checking if there is no conflict then user C sends the successful commit message to user A. Then user A will replicate this version to all the participant of the project so that everyone can get the latest version which can help to avoid more complications in future. These complications can be caused by the usage of old version.

For instance, if the participants are offline and another user had made some changes than the users who are offline will get a message, whenever they will be online again with the same project.

Otherwise, when users will start working with the project, then before they start to work, the latest version information regarding the project is gathered. If the version number is the same as the one already stored locally, then they can start working on the project. If the version information does not match with the existing version, then a message is shown to the user that the document has been changed and the current version is available to start work on.

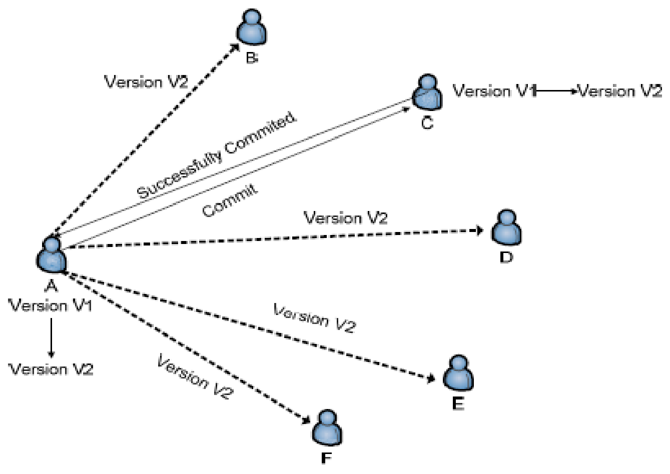


FIGURE 4.1: P2P Version Control Scenario

4.1.5 Workspace (Repository)

In most of the cases, the copies of the repository are distributed among the peers but this approach is not very helpful. In our case, the repository itself is distributed among the peers that are related to the repository anyhow. Each project and its revision form a repository object. So this is how the repository is shared between the peers. All the files have companion files those are Meta data files and are used to describe the resource in question.

4.1.6 Failsafe persistence

As each peer can leave and enter the system anytime. So that can affect the robustness of the system. Normally, this is handled by the P2p protocol used. However, we can also make our own approach on the basis that how many peers should be available most of the time and on how many peers, we can store the replicas. Generally mostly P2p systems provide very good mechanism to replicate the content but if somehow it is not fulfilling the needs than we can use the second approach that can definitely increase the robustness of the project

4.1.7 Advantages and disadvantages of P2P version control system using LIKIR

Advantages:

- Provides scalability
- Content is always online

- multiple data sources
- Robustness
- data stored encrypted
- users can work even though they are offline(automatically transmitted to other user when comes offline)
- Transmition of updates are fast and take less bandwidth
- Securely store and transfer the data

Disadvantages:

- Single point of failure (not always)
- PK management is overhead

4.2 Motivations and challenges of p2p version control system

4.2.1 Motivations

Motivations are the strong reasons behind the development of any system. To develop a p2p version control system, we also have different motivations and are discussed as under,

- Unsecure data storage.
- Vulnerable to many known attacks.
- Too much bandwidth utilization.
- Unsecure data transfer.
- No better conflict management
- Nodes are not certified.
- No node authentication.

On the basis of these motivations, we have proposed the functional specifications of a new p2p version control system which almost remove all the problems caused by available version control systems.

4.2.2 Challenges

Every p2p system comes with its on challenges so here we will see the different challenges faced by proposed version control system on the basis of LIKIR,

- certification service is single point of failure (Not always)
- PK management introduces overhead
- Better conflict management

4.3 Architectural Design

The architectural design depend upon several layers that include,

- Presentation layer
- Storage layer (Data Structure, Storage, versioning)
- Peer-to-Peer Layer

So the overall architecture of the system can be seen in Figure 4.2.

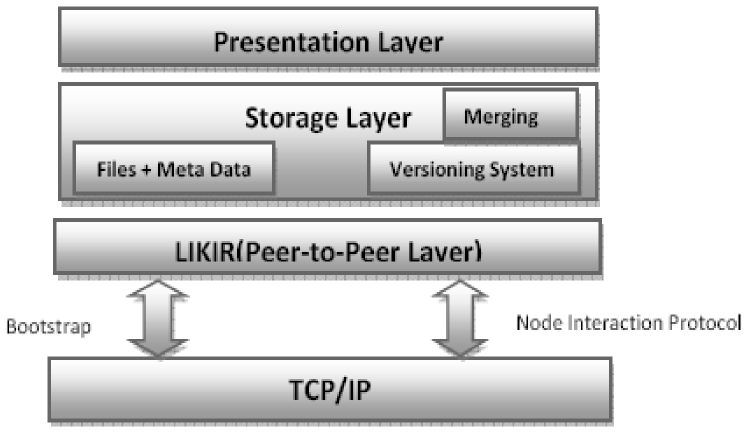


FIGURE 4.2: P2P version control Architecture

4.3.1 Peer-to-peer layer

This layer provides the low level services like peer lookup, message routing and data manipulation. So with the help of this layer a distributed network is created by using the Distributed hash tables. Data is stored on the network with a key so in this kind of network stored value can be accessed with help of a key which is generated randomly on the time of storage.

In our case, we are using the LIKIR, which provides us secure communication because each node is authenticated with the certification service. This insures the authenticity of the peer which makes network secure and removes many possible attacks that were very vulnerable to the Kademlia network.

4.3.2 Storage Layer

Storage layer includes a couple of modules, files, their Meta data and versioning. As most peer to peer systems are read only, means object stored cannot be changed again. However, in our case it’s not read only.i.e., we can make changes. So it uses less space and provides fast and efficient access and storage of files in the repository.

To make this happen and further to provide support for different file systems, we need to build a virtual file system because we don’t want to restrict our repository to a specific file system.

4.3.2.1 Virtual file system

Virtual file system helps users to manage their data efficiently. The basic features of a virtual file system are:

- Creating file/directory
- Moving file/directory
- Renaming file/directory
- Deleting file/directory
- Support different file Systems(Multiplatform)

The virtual file system with all the basic features also provides us platforms independence. It helps the users to use any operating system they want because virtual file systems can manage different file systems automatically. For this we also have to proposed a virtual file system for file storage and the proposed data structure for the virtual file system is as shown in the Figure 4.3.

All the project files have the Meta data which give description about the project files. Directories also have the metadata file which contains the information about the directories itself. Meta data is stored at the head of each file and directory that includes the information about document and the directory respectively. As we know that the Meta data consists of different attributes, or elements necessary to describe the resource. So, in our case, we use Metadata to describe the object stored. It includes owner, creation time, length, revision number, user names that are using the same document etc.

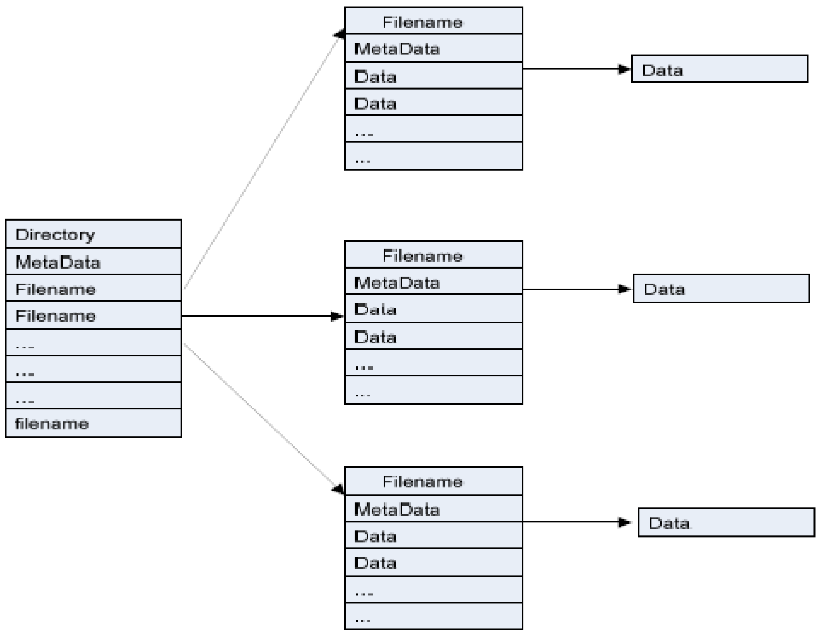


FIGURE 4.3: P2P version control Architecture

4.3.2.2 Metadata

In this case Dublin core [42] metadata will be used. It is widely used metadata to describe different kinds of content available online. The metadata set of the Dublin core consists of 15 elements that include Title, Creator, Subject, Description, Publisher, Contributor, Date, Type, Format, Identifier, Source, Language, Relation, Coverage, Rights. This element set is developed according to the general attributes of the content in our case it will not help us totally so for that we have to add more elements in the data set. (It is the same approach used in one of the papers in the related study section but with different parameters) These elements are version number, no. of participants, branching, last changed date, last changed by, base version number, total revisions and one of the most important the access control.

The version number will be used to give numbers after each successful committed operation. Version numbers are assigned in increasing order like 1223, 1224, 1455 etc. Number of participants describe that how many users are sharing the same project so whenever there will be any update regarding the project, it will be sent to all the participants to avoid future conflicts due to old versions. We can also restrict the number of users that can join the particular project.

This option depends upon the user, that how he wants to make the system work. Branching can be used if any other department is using the same project but with some changes, to manage both departments we can use branching. Last changed date describes the date when the document was last changed, and it also includes the change dates of all the users. Last changed by includes the ID of the person who edited the document, and also includes all the names of the persons who edited the files. Base version number, as the name suggests, describes the version number of the file when it was checked out and total revisions describe the number of all the revisions of the file. In access control, user access is checked whether the user who is requesting the project is eligible to access the project or not. If he is eligible access his privileges are checked to see whether he can only read the project or can do read and copy.

4.3.2.3 Merging

When more than two users are working on to the same document and one of them commits the operations that were performed on to the file then there is a possibility that conflict may occur on to the other user end. So if this happens then there is a need to identify that conflict and resolve that conflict. There are a couple of approaches to resolve these conflicts. You can either do it manually or automatically. Most of the version control softwares use the manual merging approach but now the trend is changing and version control softwares are started using the automatic approach to identify conflicts and then resolve them. However these automatic merging approaches also restricted and in some situations user has to use the manual merging.

Techniques of Merging:

There are many techniques that are proposed, we can also describe these techniques as level of merging. Because these techniques were proposed in order to make merging more and more specific. Currently available version management softwares do not support all the techniques and are described as under,

- Textual merging
- Syntactic merging
- Semantic merging

Textual Merging: In textual merging each software artifact or file is considered as text file. Currently, many version control softwares like RCS, CVS etc are using textual merging technique. The approach which is actually being used is the line based merging. So with the line based merging common lines can be detected also the new inserted, deleted and modified text lines can be detected as well. This approach is very general and cannot help to resolve all the conflicts. In spite of its disadvantages, line-based merging remains a very useful technique because of its efficiency, scalability and accuracy [47].

Syntactic merging: Syntactic merging [47] is an approach that deals with the syntax of the software artifact so that makes it more powerful than the textual merging. Textual based merging normally show many unimportant things as a conflict like code comment, and also it shows conflicts if a line is introduced or some tabs introduced, which makes code more readable.

Syntactic merging is of two types graph based and tree based. In a graph based technique the data is parsed in a graph based data structure and in the tree based technique the data is parsed in a tree based data structure.

Semantic merging: In semantic merging, different software artifacts are compared semantically to see whether there is a conflict or not. In semantic merging each version is understood according to its behavior if behavior differs in anyway then there will be a chance of conflict. For instance, let see this condition if (n>0) then n=n+n and the other one is while (n=0); n=n+n; n–; both the conditions are different but the end result is same. So there will be no conflict in it and the document can be merged automatically. If there is a difference of result then there will be a conflict.

4.3.2.4 Types of Software Merging Techniques

There are many software merging approaches that exists. Normally they can be classified into two major categories and are as under,

- State based merging [47]
- Operation based merging [48],[47]

State based Merging: In a state based merging approach only the initial and the final stages are considered when user commits changes and sends the edited file and the base version number. The base version number is the number when the user checked out the software artifact from the repository. Then that file is compared with the base version to see if the initial state of the document has been changed or not. A new version is created and stored in the repository if the state of the document has been changed and if there is any conflict than new version is not created and the users have to manually solve the problem. This approach is not very popular nowadays due to its deficiencies that include,

- It cannot accommodate multiple changes in the same line.
- Program indentation can cause the change in state of many lines which normally means nothing.

- Merging can be only applied to a limited set of types.
- State based merging tools can not provide any help in the event of any conflict.

This approach is only suitable for source code or other documents, but it is unsuitable for the binary files. Because as it deals programs as a text files when it will be applied to binary files then the repository will grow faster because each revision will produce a new whole file and then also it will require more bandwidth to transfer file over the network for comparison or to store the file in the repository. This approach is being used by the CVS, source safe etc.

Operation Base Merging: In our work, we will use the operation based approach because it is appropriate for all kinds of files. In operation based approach, when the modified object is committed the changes or the operations are committed. Operations are actually the sequence of command histories, or we can say the operation that is performed on a particular artifact. Conflicts are detected by comparing the command history or the modification operations that have been applied by different developer in parallel. Operation based merging can be used for detecting syntactic, structural and even some kinds of semantic conflicts as well [47]. Operation based merging supports the merging for all object types not for the some specific class of objects. Operation based merging resolve most of the conflicts automatically without user intervention but user intervention cannot be removed altogether at some stage user intervention is required to resolve the conflict. Currently, all the version control systems are asynchronous that means when any operation is performed on to the software artifact that is not transmitted immediately to the other end. It stores the command histories and when you wanted to commit the software artifact than all the command histories are sent to make a new version if there is no conflict. In case of conflict operation based merging sends users a specific message regarding the conflict because it has the command histories of users. The users then can resolve the conflict manually.

In [49] and [50] an asynchronous and synchronous collaboration mode and log compression are discussed respectively. Regarding an asynchronous and synchronous mode if we can use this approach in our project then it can enhance the functionality of the project, but still it’s optional. The reason is that, in many situations, it is required to switch between asynchronous to a synchronous collaboration mode. Currently, all the version control softwares are using an asynchronous mode so it will be very useful feature if it can be used.

Also sometimes logs become very large and then need large bandwidth to transfer over the network. It is possible if you are using a high bandwidth connection, but if you are using dialup or some other low bandwidth network then it will be very difficult to send logs over the network. At time, we can use log compression to utilize the network bandwidth wisely.

4.3.3 Presentation layer

Presentation layer used to present the data so it will be an interface that can be used to see the files that are checked out from the Repository. We have to create a GUI client to access all the required information.

4.4 Functional Specifications

In this section, we will describe the protocol primitives of the P2P version control system that is widely under research now days. Every LIKIR node has to be registered with the CS (Certification Service) to get a certificate to prove that the node is authentic, and it can part for any kind of transaction. This is the general description of the protocol, and it consists on a number of steps and the first couple of steps are the same as in the LIKIR protocol which includes Initialization, Join and node interaction (which we have described in previous chapters). These are the steps that are very important for each node without these steps a proper overlay network cannot be initiated. The other steps that are necessary for the P2P version control system include Project creation, project search criteria, Check out, offline commits, operation based merging, Storage criteria,

For all these operations to happen in we have to create the RPC for that. It means we have to create RPC's for P2P version control system.

4.4.1 Project Creation: (Access control)

After doing the initial steps the next step will be the creation of project. To create a project user/peer has to fill in all the attributes discussed in meta-data section earlier in order to save the project successfully. All the attributes have their own advantages that provide user the control over the content/project that the user is creating.

In these attributes, most of the attributes are given value by default but there are a couple of attributes that need to be filled in manually, which are project title, access control, no. of user's(that can provide some advantage regarding the complexity of changes made by many user's). Especially the access control and the number of users depend upon the user that how many users he wants to allow and at what level, he wants them to allow. It means that whether he wants other users to join the project, or just giving permission to just read or both read write. Furthermore, we can check the user based on a particular domain name that if the user belongs to this domain then he can join the project or can do other operations on to the project.

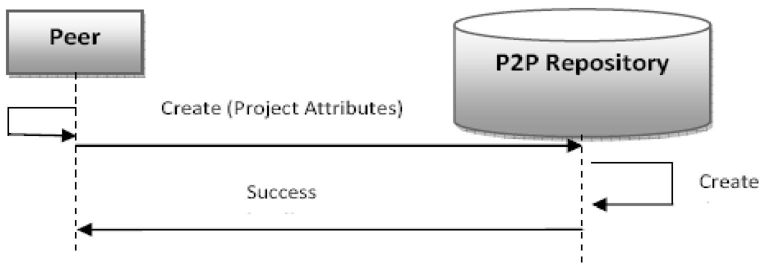


Figure 4.4: creation of the project

4.4.2 Project Search:

After the project creation, the next step is to search the specific project from the repository. It can be done in a couple of ways, already created projects can be searched with the title of the project. The project can be searched as well with the name of the user who created the project. When any peer search the project with the user name, in result he will get a list of projects that are created by the user. These are the two basic approaches that are used to search the projects.

4.4.3 Checkout

In check out the user search the project from the repository and then select the appropriate project he wants to check out. The whole procedure is as shown in the figure 4.5. In order to make this happen, we have to create an RPC for the P2P version control system. This RPC will work in two steps 1st it will search the project and show the list to the user and 2nd than creating a TCP session with a particular peer to copy the selected project.

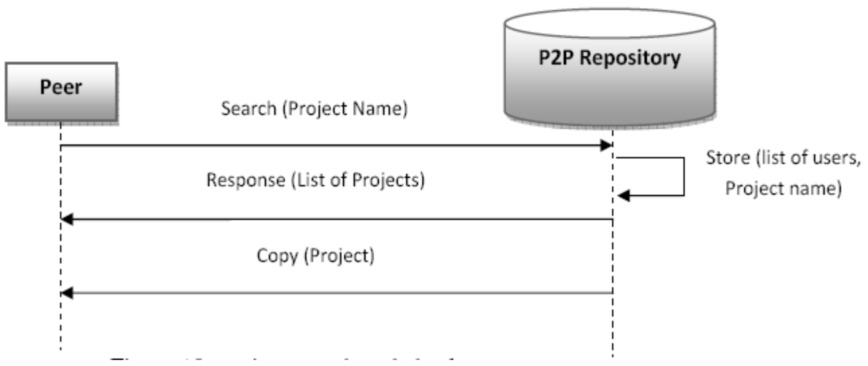

Peer
P2P Repository
Search (Project Name)
Response (List of Projects)
Store (list of users, Project name)
Copy (Project)

Figure 4.5: Project search and check out

4.4.4 Off-line Commits

This functionality is a bit same as provided by already available client server version management softwares but it is different in a way that it is for P2P network architecture, it not only commits when it sees the internet connection, it also checks for new versions before committing already made changes. In offline commits users make changes in projects and then commit these changes, even if the peers are offline. We have two problems here to solve, one how we can commit when a user comes online again and the second how the other users will get to know that there is a new version available.

To solve it, a peer needs internet connection to commit the changes. In actually, we will implement a mechanism which will automatically commit the project when the peers come online. We will create a new recursive RPC which will save the data locally if there is no internet connection is available, for the moment. The RPC will check for the internet connection after regular interval. When the peer founds the internet connection then the first thing what it will do, it will check for the new versions of the same project if it is already available then it will merge the new version automatically if it can. Otherwise if there are any collisions that cannot be solved automatically then peer has to solve the problem manually. When this thing will be done then the offline commit will be committed in actually the way ordinary commit has been done.

4.4.5 Operation based merging/Commits (also include concurrent commit)

When user made some changes in the project then user commits that project and then that project is sent to the other person who is working on to the same project. Then there we use the Operation based margining in which changes from both the users of the same project are merged and this merging can occur in two ways as discussed earlier in automatic merge and the manual merge. We can see the detail process as shown in the figure 4.6.

4.4.6 Storage Criteria

When a new version is created the next phase is to store the version to other remote machines in order to provide. To implement this process, we will have to use the default Store RPC of the Kademlia protocol. For P2P version control the specific attributes related to Kademlia can be changed like the amount of data to store and the no. of nodes on which that data can be stored. These attribute can really improve the performance of Kademlia network.

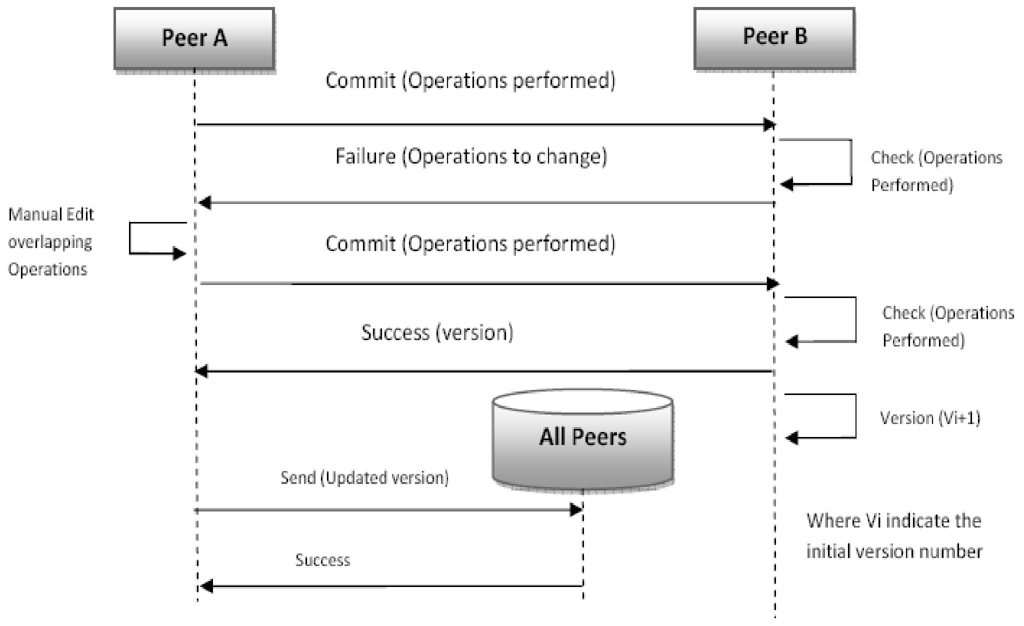

Peer A
Peer B
Commit (Operations performed)
Failure (Operations to change)
Check (Operations Performed)
Manual Edit overlapping Operations
Commit (Operations performed)
Check (Operations Performed)
Success (version)
Version (Vi+1)
All Peers
Send (Updated version)
Where Vi indicate the initial version number
Success

FIGURE 4.6: Commit and concurrent commit operations

4.4.7 Class Diagram

P2P VCS (P2p version control system) at the start can be structured into 7 packages and that includes different classes in it. These packages can be described as under,

P2pVc.p2pVC. This package is the responsible to run the application and in the background it registers the node with the certification service (CS).

P2pVc.io. This package will handle the main functionality of the P2P version control system. It's responsible for all the Remote procedure calls that are required to make P2P version control system running.

P2pVc.ui. It includes all the geographical user interfaces that will be used in during the project.

P2pVc.exceptions. It includes classes which deal with different exceptions and those exceptions are thrown when the application cannot be started normally and if something went wrong during the running of application.

P2pVc.security. It provides a random number generator (the Mersenne-Twister) and a Security Suite class. The latter keeps the user DES key, generated by the password provided during login and exposes methods to encrypt and decipher the content.

P2pVc.settings. It contains two simple classes to retrieve settings from the config.properties file. The PropFinder class provides an interface to access to properties.

P2pVc.util. It includes only the Content Manager class that provides static methods for Object serialization and de-serialization.

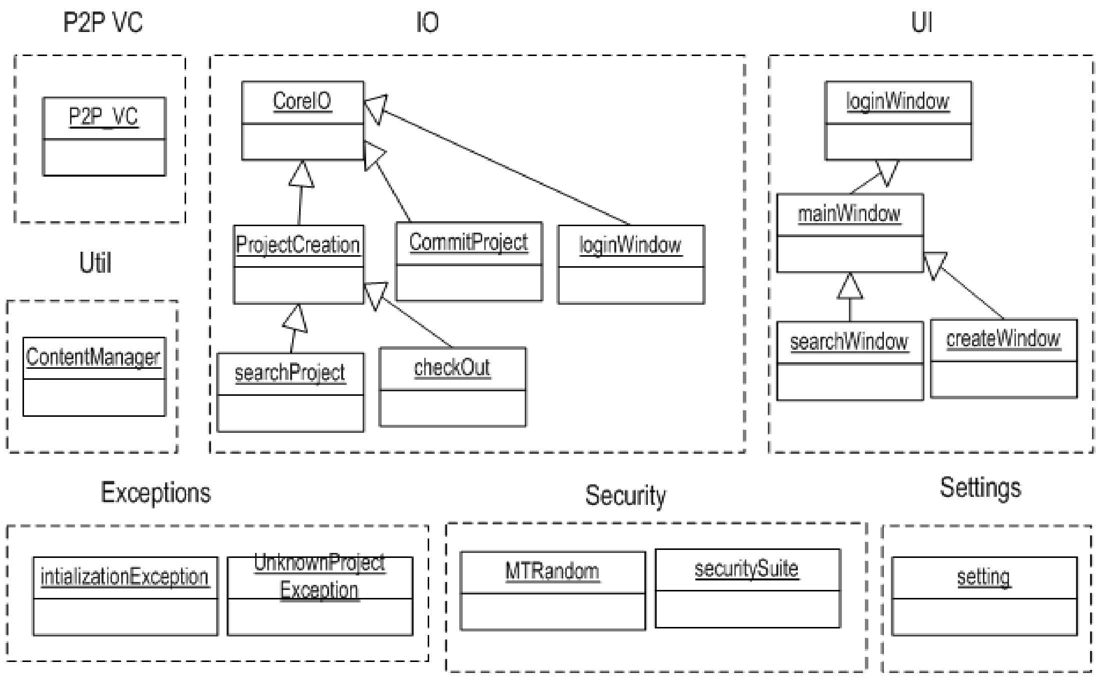


FIGURE 4.7: Model Class diagram

4.5 Analysis of previous design approaches

In this analysis, we will only consider [4] and [5]. As they are the only available p2p version control systems and presents good understanding of the basic requirements and design of version control system. But they are still not mature and lot more work is needed on them. There research is being conducted continuously to make these systems work in a practical environment.

4.5.1 Distributed Hash Table Based Peer-to-Peer Version Control System for Collaboration [4]

Yi Jiang, Guangtao Xue, Jinyuan You provide a good design of DHT based p2p version control system. The design was based on three main goals and as under,

- System should support concurrent editing, read-modify-write style operations on objects and also better software merging.
- system should support merits of p2p architecture that includes locality transparency and mobile user support.

- There should be a simple geographical user interface providing basic functionalities like, lookup(searching functionality), put(saving content), get(retrieving content) and versioning of the content.

This design provides all the basic features that are necessary for any version control system like checkout, commit, concurrent commit, automatic merging, versioning and distributed repositroy.

Advantages:

- Purely peer-to-peer
- Automatic merging
- Concurrent commit
- Read-write file system
- Low bandwidth requirement
- Only file containing operations is sent instead of whole file

Disadvantages:

- Vulnerable to many attacks
- Data is not stored in encrypted form
- Unsecure data transfer
- No client authentication
- Nodes are not certified
- No failsafe persistence
- no offline commits

4.5.2 Peer-to-Peer based Version Control [5]

Patrick Mukherjee, Christof Leng, Wesley W. Terpstra, Andy Schurr provide another design of p2p version control system. They have used FreePastry as an underlaying p2p protocol but other protocol can also be used. It also provides all the basic functions that are required by a p2p version control system and it includes, Repository distribution, fail-safe persistence, checkout, commit and concurrent commits.

Advantages:

- Purely Peer-to-Peer
- Automatic merging
- Concurrent commits
- Read-write file system
- Low bandwidth requirements
- Only file containing operations is sent instead of whole file

Disadvantages:

- No automatic merging
- Vulnerable to many attacks
- Data is not stored in encrypted form
- Unsecure data transfer
- No client authentication
- Nodes are not certified
- no offline commits

4.5.3 Comparison of design approaches [56]

The comparison criteria is based on a number of properties like, Conflict resolution, Model, Units of operation, Concurrent commit, Data model, Container identity, File renames, offline commits, failsafe persistence and security. We will use these parameters to analyze the properties of different of different design alternatives.

4.5.3.1 Conflict resolution

We have already discussed about the two techniques for conflict resolution one of them is operation based merging and the other is state based merging. In this parameter we will see which one is being used.

4.5.3.2 Model

In model, we will see which type of architecture they are based on that can be peer-to-peer or distributed as well. Actually they all are peer-to-peer based but it was necessary to show here every one can understand easily.

4.5.3.3 Units of operation

In units of operation, we have two kinds of operations file-based and file-sets based file based operations were used in early version control systems like CVS, RCS etc and some of which are also being used today. In file-based operations every file has its own master file and comment related to it. But in file-set based operations change set is considered as a unit by the application and any comment associated with change does not belong to any one file.

4.5.3.4 Concurrent commits

In concurrent commits, more than one user can make changes on a file and can commit at the same time is called concurrent commit. Some systems support this kind of property and other don't support.

4.5.3.5 Data model

In data model, we have two kinds of approaches on is snapshot based and the other is change set based. These are used to maintain the history of line of development. Both techniques are being used but currently change set based technique is being used.

4.5.3.6 Container identity

It is used to assign files and directories a stable internal identity. It is very helpful in case file, directory renames and moves. Normally, it is being used in all the new version control systems.

4.5.3.7 offline commits

In offline commits users can vitually commit even when they are not connect to the system. But next time when they will connect to the system all the offline commits are send to all the participants nodes in the project.

Table 4.1: Comparison between design approaches

Properties	DHT based P2P VCS	P2P based VCS	Proposed System
Conflict Resolution	Operation Based	Operation based	Manually(not Operation based)
Model	Peer-to-peer	Peer-to-peer	Peer-to-peer
Units of Operation	Filesets	Filesets	Filesets
Concurrent commit	Yes	Yes	Yes
Data model	Changeset	Changeset	Changeset
Container Identity	Yes	Yes	Yes
offline commits	Yes	Yes	No
failsafe persistence	Yes	No	Yes
Security	Yes	No	No

4.5.3.8 failsafe persistence

In failsafe persistence is used to manage the availability and robustness of the system by controlling the number of peers that can save data when the commit request is send to make data available all the time.

4.5.3.9 Security

None of the available design alternatives and the other system provide security. In security, we are considers the following points,

- Node Authentication
- Node certification
- Encryption of data
- can sustain present attacks of p2p environment.

4.6 Application programming interfaces

To develop this protocol we need different kind of application programming interfaces and these application programming interfaces used to provide different kind of functionalities which are necessary for the development of this protocol architecture. Functionalities that are required for the development of this protocol architecture are secure P2P overlay architecture; operation based merging API to merge changes automatically made by different users in different documents and to make things secure in a view of encryption and to generate random numbers to give user's random user ID's. prototype

All these details are discussed earlier in different sections of this chapter. In this protocol architecture, there will be many application programming interfaces that can be used to develop a P2P version control system and are described as under,

4.6.1 LIKIR

LIKIR will be the main API which will be used in order to create a secure P2P overlay network because lack of authentication in P2P networks leads to a number of attacks that can ultimately harm the network to a maximum level. It's a framework that includes an identity based scheme and a secure communication protocol, built on top of Kademlia.

4.6.2 Operation based merging API

As we are using the operation based merging and for this purpose we can use an eclipse plug-in which is specially made for operation based merging and its name is MolhadoRef [51]. It is an object oriented SCM infrastructure which uses eclipse as a front end. It provides all the functions required by P2P version control architecture they are not exactly the same but obviously they can be modified according to the needs and can be used afterwards.

4.6.3 Security

To provide security at content storage level in the network. For that we need to encrypt the data when it is being stored on the network and decrypt the data when we are recover the data from the network. For that we will use Bouncy Castle crypto API [52] which will provide us with light weight cryptography technique. So it will be used to encrypt and decrypt content that we are going to store in the repository.

Chapter 5

Conclusion and future work

This chapter present s the conclusion and the future work of the this thesis work and is as follows,

5.1 Conclusion

In this thesis work our focus was on p2p networking specifically study of distributed hash tables and p2p version control systems. This thesis work is a combination of different tasks that include LIKIR, LiCha analysis, LiCha module development and analyzing the possibility of p2p version control system and afterwards proposing the functional specifications of p2p version control system.

In LIKIR, LiCha analysis part, we have done the analysis of LIKIR and LiCha by keeping in mind that we had to introduce some modifications in the previous work done. During the analysis, we have proposed some modifications for both LIKIR and LiCha. In case of LIKIR, we have proposed a user information module which is a standalone module and can be very helpful in many applications which uses user information for many purposes as discussed earlier. In this module applications can get user information at any stage of the application life cycle and afterwards use this information to provide better services, advertisements(content management services) etc. In case of LiCha, we have proposed many solutions out of which we have selected the combined chat module and it is implemented as planned and evaluated using the set of goals that were defined prior to the implementation of the combined chat module. In combined chat module initially user can only select up to a defined number of contacts. This chat application performs all the basic functions the way any other p2p application can perform like Skype. In case of p2p version control, at first we have studied available p2p version control systems and afterwards we saw the possibilities for developing p2p version control system on top of LIKIR keeping in mind all the basic functionalities that should be in p2p version control system. In the end, functional specifications of p2p version control system were proposed using the best practices available.

This thesis work gives great insight of p2p networking, information about different distributed hash tables, traditional version control systems and pure p2p version control systems which are very rare these days.

5.2 Future work

There is a lot of tendency for improvements, in case of LIKIR and in this thesis work as well. As LIKIR is newly built protocol so I personally believe that it will take some time and more testing on the basis of industrial standards to make it perfect so that it can be used to build professional applications.

In case of this thesis work there can be many things that can be improved and new things can be also be introduced in future. As this thesis work is a combination of different tasks, In case of LIKIR, LiCha analysis we have proposed user information module for LIKIR that can be used to gather user information or else it can be used whenever it is needed. UIM module can be developed in future as it has many applications in different areas as already discussed briefly. In case of LiCha, we already proposed different improvement that can make LiCha a more professional application which is just a basic chat application at the moment (more specifically a prototype). LiCha can have a bright scope as p2p chat application which is open source and will provide all the functionalities that any existing application provides if the proper amount of work can be done to improve LiCha.

In combined chat application, only a number of limited peers can be invited for combined chat, which was consider one of the requirements at the start of the application. It can be improved by modifying the code and then afterwards as many contacts as a peer wants can be added in combined chat. With this addition, we can use the same code to develop chat rooms as well. The chat rooms can be of different types depending upon the interests of the users.

Last but not least, this thesis work presents functional specifications and some guidelines that can help researchers in future to implement this system with minimum possible difficulties. As there is lot of work going on at p2p networking. Still there is no pure p2p version control system that is available at present and can be used. So with the development of this system hopefully there will be big change in the industry of version control system.

Appendix A

Source code

All the source code of the LiCha combined chat is now available. We have created a project on assembla code and uploaded all the source code on it with the consent of Renato Lo Cigno (University of Trento, Italy). Now code accessibility is very easy for those every one who wants to access the code and make changes in it. To access all the details of the LiCha combined chat project page please visit Assembla home.

Bibliography

[1] L. M. Aiello, M. Milanesio, G. Ruffo, R.Schifanella ”Tempering Kademlia with a Robust Identity Based System”, In the 8th International Conference on Peer-to-Peer Computing 2008 (P2P’08), September 8-11, 2008, RWTH Aachen University, Germany. IEEE Press.

[2] P. Maymounkov and D. Mazi‘eres. Kademlia: A peer-to-peer information system based on the xor metric. In IPTPS 2002, pages 53-65, 2002.

[3] Edoardo Rossi, ”LICHA: applicazione decentralizzata di communicaione ”Instant Messaging” resistente ad attacchi su rete P2p” Thesis in Facolt di Scienze Matematiche Fisiche e Naturali, University of Torino, 2007/2008

[4] Yi Jiang, Guangtao Xue, Jinyuan You ”Distributed Hash Table Based Peer-to-Peer Version Control System for Collaboration”. CSCWD 2006: 489-498

[5] Patrick Mukherjee, Christof Leng, Wesley W. Terpstra, Andy Schrr, ”Peer-to-Peer Based Version Control,” icpads, pp.829-834, 2008 14th IEEE International Conference on Parallel and Distributed Systems, 2008.

[6] Rdiger Schollmeier, A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications, Proceedings of the First International Conference on Peer-to-Peer Computing, IEEE (2002)

[7] Andy, Oram. Peer to Peer: Harnessing the Power of Disruptive Technologies. O’Reilly & Associates, Inc, 2001

[8] ipoque available at http://www.ipoque.com/lastvisited:23November,2009.

[9] Ipoque internet study 2007, available at http://www.ipoque.com/resources/internet-studies/internet-study-2007 Last visited: 23 November, 2009.

[10] Ipoque internet study 2008/09, available at http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009 Last visited: 23 November, 2009.

[11] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In Proc. HotOS VIII, Schloss Elmau, Germany, May 2001.

[12] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In Proc. ACM SOSP'01, Banff, Canada, Oct. 2001.

[13] A. Rowstron, A.-M. Kermarrec, P. Druschel, and M. Castro. Scribe: The design of a large-scale event notification infrastructure. Submitted for publication. June 2001. havailable at `ttp://www.research.microsoft.com/antr/SCRIBE/`.

[14] Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In Proceedings of SIGCOMM 2001, San Deigo, CA, August 2001.

[15] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized ObjectLocation, and Routing for Large-Scale Peer-to-Peer Systems. Proc. IFIP/ACM Middleware 2001.

[16] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment, Appears in IEEE Journal on Selected Areas in Communications, Vol 22, No. 1, January 2004.

[17] Gnutella, available at `http://rfc-gnutella.sourceforge.net/` Last visited: 24 November, 2009.

[18] eMule, available at `http://www.emule-project.net/` Last visited: 24 November, 2009

[19] Cederqvist, P., et al.: Version Management with CVS. Manual for CVS available at `http://ftp.gnu.org/non-gnu/cvs/source/stable/1.11.23/cederqvist-1.11.23.pdf` Last visited 04 December, 2009

[20] Ticky, W.F.: RCS - A system for version control. Software Practice and Experience 15(7) (1985) 637-654.

[21] Subversion, available at `http://subversion.tigris.org/` , Last Visited: June 1, 2009

[22] Microsoft Visual SourceSafe. available at `http://msdn.microsoft.com/enus/library/aa302175.aspx`, Last visited: June 1, 2009

[23] KDE, available at `http://www.kde.org/` Last visited: 24 November, 2009

[24] Gnome, available at`http://www.gnome.org/` Last visited 24 November, 2009

[25] Mozilla available at `http://www.mozilla.org/` Last visited 24 November, 2009

[26] chy, W. F. 1985. RCS-a system for version control. Softw. Pract. Exper. 15, 7 (Jul. 1985), 637-654. available at `http://dx.doi.org/10.1002/spe.4380150703` Last visited: 04 December. 2009

[27] apache, available at `http://www.apache.org/` Last visited: 24 November, 2009

[28] FreePascal, available athttp://www.freepascal.org/ Last visited: 24 November, 2009

[29] FreeBSD, available at http://www.freebsd.org/ Last visited: 24 November, 2009

[30] GCC, available at http://gcc.gnu.org/ Last visited: 24 November, 2009

[31] Python, available at http://www.python.org/ Last Visited: 24 November, 2009

[32] Google code, available athttp://code.google.com/ Last visited: 24 November, 2009

[33] Django, http://www.djangoproject.com/ Last visited: 24 November, 2009

[34] Ruby, available at http://www.ruby-lang.org/en/ Last visited: 24 November, 2009

[35] GIT, The fast version Control System, available athttp://git-scm.com/ last Visited: June 1, 2009.

[36] GNU arch, available at http://www.gnu.org/software/gnu-arch/ last visited: June 1, 2009.

[37] Monotone, available at http://www.monotone.ca/ last Visited: June 1, 2009.

[38] Mercurial, available at http://www.selenic.com/mercurial/wiki/ last visited: June 1, 2009.

[39] Wine, available at http://www.winehq.org/ Last Visited: 24 November, 2009

[40] D. Dumitriu, E. Knightly, A. Kuzmanovic, I. Stoica, and W. Zwaenepoel. Denial-of-service resilience in peer-to-peer file sharing systems. SIGMETRICS Perform. Eval. Rev.,33(1):38-49, 2005.

[41] M. S. et al. A global view of kad. In IMC '07: Proc. of the 7th ACM SIGCOMM, pages 117-122, New York, NY, USA, 2007. ACM.

[42] Dublin Core Metadata. URL: http://dublincore.org, Last Visited at: 20 May, 2009

[43] Mens, T.: A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering 28(5) (2002) 449-462

[44] Lippe, E. and van Oosterom, N.: Operation-Based Merging. ACM SIGSOFT Software Eng. Notes 17(5) (1992) 78-87

[45] FreePastry, available athttp://www.freepastry.org/FreePastry/ last visited: 26 November 2009

[46] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 (1998).

[47] Mens, T.: A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering 28(5) (2002) 449-462

[48] Lippe, E. and van Oosterom, N.: Operation-Based Merging. ACM SIGSOFT Software Eng. Notes 17(5) (1992) 78-87

[49] Haifeng Shen and Chengzheng Sun: "Operation-based revision control systems" School of Computing and Information Technology, Griffith University, Austerlia

[50] Shen, Haifeng and Sun, Chengzheng, "A Log Compression Algorithm for Operation-based Version Control Systems", COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment, year 2002,pages 867-872

[56] MolhadoRef: Refactoring-aware SCM system compression `https://netfiles.uiuc.edu/dig/MolhadoRef/molhadoref.html` last Visited: 22 July 2009.

[52] Bouncy Castle crypto API available at `http://www.bouncycastle.org/latest_releases.html` last Visited: 22 July 2009.

[53] Skype, available at `http://www.skype.com` last visited: 1 December, 2009

[54] Peer-to-peer, available at `http://en.wikipedia.org/wiki/Peer-to-peer` last visited: 22 April, 2010

[55] Beverly Yang, Hector Garcia-Molina, "Designing a Super-peer Network." In Proceedings of the 19th International Conference on Data Engineering (ICDE), Bangalore, India, March 2003

[56] Eric Raymond: Understanding Version-Control Systems (DRAFT) `http://www.catb.org/~esr/writings/version-control/version-control.html` last Visited: 19 May, 2009.