ABSTRACT

MURTHY, PRASHANT. GTKgREP - Design and Implementation of a Gnutella-based Reputation Management System. (Under the direction of Dr. Douglas Reeves).

Peer to peer (P2P) networks have introduced a new paradigm in content distribution. Such systems have shifted the paradigm from a client-server model into a client-client model. The tremendous success of such systems has proven that purely distributed search systems are feasible and that they may change the way we interact on the Internet. Most P2P protocols have been designed with minimum or no emphasis on security – Gnutella being one such open protocol standard. In this work, we focus on providing security over Gnutella by establishing trust between the entities (peers) in a P2P network using reputations and by ensuring integrity, authentication and non-repudiation of messages exchanged. Reputation systems collect, distribute and aggregate feedback about past behavior of the participants. Such systems help in establishing trust amongst strangers, detecting misbehaving nodes and isolating them. In this work, we analyze some existing reputation-based protocols in P2P networks. Among these protocols, we choose two approaches that are more specific and relevant to P2P networks. We compare these two protocols, namely, *P2PRep* and *RCert* in terms of security and performance. While P2PRep uses a broadcast polling mechanism and client-side storage to manage reputations, RCert uses unicast messages and server-side (local) storage of reputation content. Based on an analysis of the two approaches, we choose to enhance *RCert*. We identify the shortcomings and vulnerabilities of this protocol and propose an extension to RCert. We then provide the details of the design and implementation of our enhanced protocol – *GTKgREP* on Gtk-Gnutella, a unix-based Gnutella servent. We provide an assessment of the overheads associated with this protocol.

GTKgREP - Design and Implementation of a Gnutella-based Reputation Management System

by

Prashant Murthy

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science

Department of Computer Science

Raleigh

2003

Approved By:

Dr. Peng Ning	Dr. Munindar Singh

Dr. Douglas Reeves
Chair of Advisory Committee

To my parents,

K S Keshava Murthy,

Lakshmi Murthy

and my sister and brother-in-law,

Sheetal Murthy, Suresh Byagathvalli

Biography

Prashant Murthy was born in Bombay, India, in 1979. He received his Bachelor's degree in Computer Science from Birla Institute of Technology and Science, India in 2000. From June 2000 to June 2001, he worked for Hughes Software Systems and Cisco Systems as a software engineer in the VoIP group. Since 2001, he has been a Masters student in the Department of Computer Science at North Carolina State University.

Acknowledgements

I acknowledge the efforts of Dr. Douglas Reeves, my thesis advisor, in providing me direction and guidance when required. He helped me remain focused and persistent. Working with him has been a pleasure and honor.

I thank Dr. Singh and Dr. Ning for serving on my thesis advisory committee and providing me with invaluable advice.

I thank my family for encouraging me to take up graduate studies, especially my sister. In this endeavor, they have put in just as much effort as me.

Among my pals, I thank Koundi for accelerating my learning curve with Latex to get me started on writing my thesis draft and Pushkin for patiently proof reading the final draft. I would like to thank my other roommates Bhasker and Naveen for everything !

Contents

List of Figures **viii**

List of Tables **ix**

1 Introduction **1**
1.1 Reputation Systems 4

2 Related Work **7**
2.1 Architectures in P2P Networks 7
2.1.1 Centralized Networks 8
2.1.2 Decentralized (Pure P2P) Networks 8
2.1.3 Hybrid Networks 8
2.2 Reputation Management 10
2.2.1 Centralized Approaches 11
2.2.2 Decentralized Approaches 11

3 Review of Gnutella Design and Implementation **15**
3.1 Operation 16
3.2 Protocol Description 16
3.3 Protocol Messages 17
3.3.1 Message Header 18
3.3.2 Message Types 19
3.4 GGEP Extensions 21
3.5 Gtk-Gnutella 22
3.5.1 Features 22
3.5.2 Design 24

4 Review and Analysis of RCert Reputation Management Protocol **25**
4.1 Introduction 25
4.2 Overview 26
4.2.1 Rating Certificates (RCert) 27
4.2.2 Recommendations 28

4.3 Protocol description 28
4.3.1 Discussion 32
4.4 Protocol enhancements 34
4.4.1 Message Extensions 34
4.4.2 Security Extensions 36
4.5 Security analysis 39
4.5.1 Integrity and Non-repudiation 40
4.5.2 Authentication of messages 40
4.5.3 Validity of RCerts 40
4.5.4 Misbehaving entities 40
4.5.5 Collusions 42
4.6 Extended RCert vs P2PRep/XRep 43
4.6.1 Security Analysis of P2PRep/XRep 43
4.6.2 Comparison of the two approaches 44

5 Design and Implementation of a Gnutella-based Reputation Management System 47
5.1 GTKgREP Subsystems 48
5.1.1 Message/Routing Subsystem 48
5.1.2 Connection Manager 48
5.1.3 Resource Manager 49
5.1.4 GUI Subsystem 50
5.1.5 HTTP Subsystem 50
5.1.6 Property and Stats Database 50
5.1.7 PKI Manager 50
5.1.8 GTKgREP Reputation Subsystem 51
5.1.9 GTKgREP Reputation Database 51
5.2 Design Issues 51
5.2.1 Stateless Protocol 51
5.2.2 Gui-Core Dependency 52
5.2.3 Non-persistent GUID 52
5.2.4 Broadcast searches 52
5.2.5 Size of search messages 52
5.3 Message Formats 53
5.4 State Machine 55
5.5 Flow of events 55
5.6 Test Scenarios 58
5.7 Assessment of overheads 61
5.7.1 Bandwidth Usage 61
5.7.2 Round Trip Delays 62
5.7.3 Cryptographic Statistics 63

6 Conclusion and Future Work 65
6.1 Future Work 66

Bibliography **67**

A Data Structures in GTKgREP **71**

List of Figures

2.1 Architectures in P2P Networks 9

3.1 Client-Server handshake in Gnutella 17

4.1 RCert Protocol Message Flow 29
4.2 Protocol behavior with offline recommenders 31
4.3 Protocol behavior when most-recent recommender is offline 33
4.4 RCert Protocol Message Flow (Modified) 35
4.5 Message Authentication in Extended RCert 37
4.6 Selective Certificate Deletions 39
4.7 Onion Signatures in RCert 39

5.1 GTKgREP Components 49
5.2 GTKgREP State Machine 56
5.3 GTKgREP Sequence Diagram 57
5.4 P2P Test Bed 59
5.5 Certificate Bandwidth vs Number of providers and recommenders 61
5.6 Total Bandwidth Consumption 62

List of Tables

3.1 Gnutella Message Header 18
3.2 Gnutella Message Types 19
3.3 Query Message Format 20
3.4 Query Hit Message Format 20
3.5 Result Set Structure 20
3.6 Push Message Format 21
3.7 GGEP Extension Format 21

4.1 Rating Certificates 27
4.2 RCert Header Format 27
4.3 RCert Unit 28
4.4 Recommendations 28

5.1 GTKgREP Request Message Format 53
5.2 GTKgREP Message Formats 54
5.3 Number of Signatures Computed/Verified 63
5.4 Total Signatures and Hashes 63

Chapter 1

Introduction

Peer-to-peer networks (P2P) have become a very popular medium of content distribution in the past few years. In such networks, computer resources and services are exchanged directly between systems. Beginning with the popular Napster [9] phenomenon in the late 1990s, the popularity of P2P has dramatically increased the volume of data transferred between Internet users. Recent studies concluded that the file sharing activity on P2P networks accounts for up to 60% of the Internet traffic on any given service provider network [26].

P2P Networks provide an alternate mechanism to client/server paradigms for sharing resources. Existing P2P systems ([4], [2], [8], [9]) have been driving a major paradigm shift in the era of distributed computing. Many factors have fostered the recent explosive growth of such networks: low cost attributed to the fact that P2P networks may not require any high-performance centralized servers, high availability of large numbers of computing and storage resources and increased network connectivity.

Many characteristics of P2P networks have made it popular for resource sharing applications[38]:

- *Ability to operate in a dynamic environment*: P2P applications operate in dynamic environments where hosts may join or leave the network frequently. The network must achieve flexibility in order to keep operating transparently despite a constantly changing set of resources.

- *Performance and Scalability*: P2P paradigm shows its full potential only on large-scale deployments where the limits of the traditional client/server paradigm become obvious. Moreover, scalability is important as P2P applications exhibit the *network effect* [28]: the value of a network to an individual user increases with the total number of users participating in the network. Ideally, when increasing the number of nodes, aggregate storage space and file availability should grow linearly, response time should remain constant, while search throughput should remain high or grow.

- *Availability of resources*: The same resource may be available at many different nodes simultaneously, thus providing high availability of these resources.

- *No single point of failure*: Since P2P networks do not depend on any centralized infrastructure, there cannot be a common point of failure.

The nodes in P2P networks are termed as servents (or peers) since they act both as a client and as a server to the network. Content retrieval in P2P networks usually involves a *content search* phase and a *content download* phase. First, the initiator searches for the desired content in the network. Then, after having discovered a servent sharing the above content, it establishes a direct connection and transfers the resource from the provider.

Peer-to-peer networks can be broadly classified into two categories: 1. Centralized networks such as Napster, 2. Decentralized networks such as Gnutella. In the former approach, content search is facilitated by a central server that maintains file indexes to resources shared on the network. In the decentralized approach, the entities directly discover each other by exchanging broadcast messages or by other mechanisms without involving any central server. More recently, a third class of networks that adopt both approaches have become prevalent. Networks such as *Kazaa* [6] distinguish their peers into leaf nodes and supernodes based on the computation power and bandwidth limitations of the servent. Leaf nodes (low computation power, low bandwidth) are usually connected only to supernodes and are not involved in routing P2P traffic. Supernodes are servents that are endowed with additional responsibilities such as routing, handling requests, etc.

The design objectives of decentralization and availability make P2P applications very appealing for deployment in large scale networks. However, the open nature of these networks makes them vulnerable and highly susceptible to security attacks. Some of the significant security issues have been summarized below:

- *Secure Routing*: Since P2P networks form a virtual network of peers at the application layer, a P2P message (or packet) sent by the source node may need to traverse many peers before reaching the destination. This implies that intermediate peers are involved in routing packets to the destination peer. Hence, there needs to be a mechanism to maintain the authenticity, confidentiality and integrity of messages. There has been some work on avoiding attacks that prevent correct message delivery in P2P networks [14].

- *Peer Authentication*: Since there is no central authority to validate the credentials of the peers in the network, authentication is a non-trivial issue. Further, the nodes involved in exchanging resources may never have interacted with each other, thus making it difficult for them to authenticate each other. This could lead to lack of accountability of shared content. Thus, the network can be extremely susceptible to attacks by malicious users who could spread harmful viruses and trojans, or waste network resources. The Mandragore virus [5] that brought down the Gnutella network in February 2001 is an example of such an attack.

- *Anonymity*: Anonymity is a property that allows a user to conceal its identity, while allowing access to the resources in the network. There have been many proposals addressing the issue of anonymity in P2P networks [13] [16] [39].

- *Accountability*: In most P2P networks, peers obtain a unique identity (pseudonym) to log into the network. However, these pseudonyms may not be persistent, thus making it difficult to address the problem of accountability of transactions. Most P2P networks suffer from free-riding [12] – a phenomenon where peers do not share any resources. It has been shown that almost 70% of Gnutella users share no files, and 50% of all responses are returned by the top 1% of sharing hosts. Further, nodes may come online and go offline in very short intervals of time; only a small percentage of the peers are online for a considerable period of time. All nodes are involved in routing request packets to the destination nodes, thus demanding cooperation from all peers.

- *Trust Management*: Trust plays a central role in many aspects of computing, especially those related to use of the network. Many of the characteristics of P2P networks make it very challenging to establish trust among the participating nodes. As described

above, there is no central authority to monitor and coordinate activities of all nodes in the network. The peers are autonomous and may be unreliable and non-cooperating. There must be a mechanism by which the network is able to distinguish trusted peers from the malicious peers. Hence, it is very essential to have a system by which peers are able to trust each other and ensure the authenticity of the shared content. There have been many different approaches that have been adopted to establish trust in a network [32] [40]. Reputation-based trust is one specific form of trust management, which is described in detail in Section 1.1

Thus, peer-to-peer networks are very useful for deployment in large-scale networks. However, for successful deployment of these networks, the above security issues must be addressed. Since trust is a fundamental security issue in these networks, it is very essential to address it and identify solutions to provide the same.

1.1 Reputation Systems

In general, reputation can be defined as the general estimation in which a person is held by the public. It is the memory and summary of behavior from past transactions. It describes the popularity of an individual as seen by the other nodes in the network [35]. In the physical world, when we talk about how much we trust someone, we often consider that person's reputation. We usually are willing to put great trust in someone whom we have personally observed to be highly capable and who has a high level of integrity. In the absence of personal observation, the recommendation of a trusted friend can lead to trust in someone. Hence, in real life, trust is often increased by establishing positive reputations and networks for conveying these reputations.

A reputation system collects, distributes and aggregates feedback about past behavior of the participants. Reputation systems help in fostering trust amongst strangers and deter participation by those who are dishonest. Such systems have existed long before the Internet came into existence. Credit and employment agencies, for example, are indeed reputation-based systems. An individual with a good credit history (reputation) is able to receive credits or loans. On the Internet, most online auction portals such as eBay [22] provide reputations in the form of feedback forums. After completing a transaction, the buyer and seller have an opportunity to rate each other (1, 0, or -1) and provide comments. Each

participant has a running total of the points attached to its identity on the portal. This information is stored and retrieved for future transactions as necessary. Thus, reputation systems seek to restore information about the history of transactions with the expectation that other people will use this as a reference in the future.

The most essential properties that are required for a reputation system at the minimum are persistence of participating entities, a mechanism for capture and distribution feedback and storage of cumulative ratings to be made available in the future. Such reputation systems can be well adapted into peer-to-peer networks where trust amongst participating nodes is very critical. However, the decentralized property and certain other characteristics of P2P networks make it challenging to adapt reputation systems directly in such networks.

In the recent past, there has been some work in establishing trust using reputations in P2P networks [25] [41] [11] [43] [27] [31] [17] [34], as discussed in Chapter 2. At the time of this work, there was no reputation-based implementation to realize the practical deployment of these systems in P2P networks. In this work, we analyze some approaches that have been proposed to address the problem of reputation based trust management. Bearing the goals of establishing trust among peers in a secure manner and the performance constraints in P2P networks, we identify two protocols more specific and relevant to P2P networks – *RCert* and *P2PRep*. We compare the two approaches in terms of security and performance (overheads). Based on an analysis of the two approaches, we conclude that both approaches are very similar in providing the required security functionalities. However, since the RCert protocol performs better in terms of overheads, we choose to enhance RCert. We identify the drawbacks and vulnerabilities of this protocol and propose an extension to RCert. We provide the design of the modified system (GTKgREP) over an existing decentralized P2P network - *Gnutella*. We provide the details of the implementation of GTKgREP that has been built as an extension to *Gtk-Gnutella*, an existing GTK-based Gnutella servent. We also demonstrate the impact on the performance in terms of bandwidth, round-trip time and processing delays due to cryptography.

The rest of this thesis is organized as follows. In the next chapter we briefly describe the context in which this thesis and other related work is relevant. Chapter 3 describes the Gnutella protocol in detail and the features provided by Gtk-gnutella, an implementation of Gnutella. In Chapter 4, we describe the RCert protocol in detail, propose extensions, identify vulnerabilities and propose solutions to safeguard the protocol against

these vulnerabilities. In this chapter, we also provide a security analysis of the protocol and compare this protocol against P2PRep and justify why we chose to implement the RCert protocol. In Chapter 5, we provide the details of the design and implementation of the extended RCert protocol over Gtk-Gnutella and provide the results. Chapter 6 concludes this thesis and discusses possible directions for future research.

Chapter 2

Related Work

In this chapter, we provide a brief review of peer-to-peer networks. We describe the architectures in P2P networks and highlight some of the reputation-based approaches that have been proposed.

Peer-to-peer networking is the utilization of the relatively powerful computers (personal computers) that exist at the edge of the Internet for more than just client-based computing tasks. Such networks provide an alternate mechanism of computing as against client/server architectures. In addition to the ability to pool together and harness large amounts of resources, the strengths of existing P2P systems ([4], [2], [8], [9]) include self-organization, load-balancing, cost-effective infrastructure, adaptation and fault tolerance. Because of these desirable qualities, many research projects have been focused on understanding the issues surrounding these systems and improving their performance [37], [30], [20].

2.1 Architectures in P2P Networks

Based on the method by which search requests are sent into the network, peer-to-peer networks can be categorized as *Centralized* Networks, *Decentralized* Networks and *Hybrid* Networks.

2.1.1 Centralized Networks

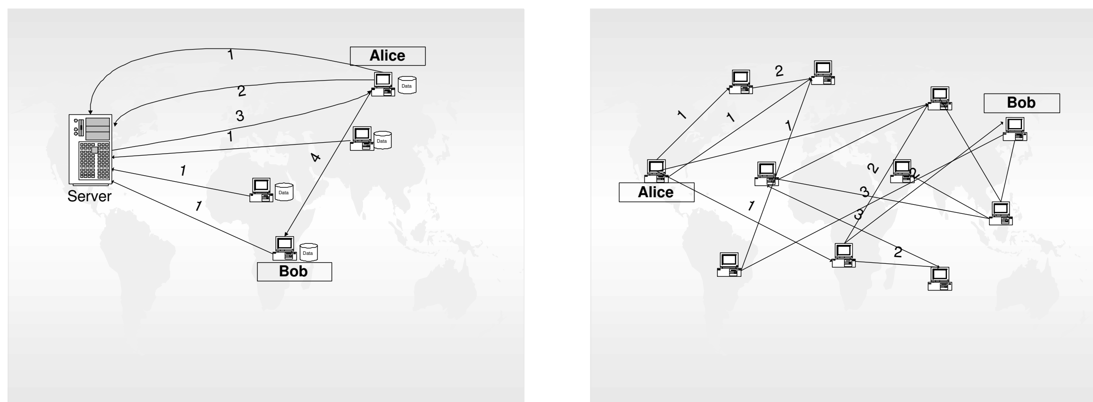
Such networks make use of a centralized indexing service. Figure 2.1(a) depicts the design of such networks and the steps involved in resource sharing. A central server stores information about the files that exist on each peer in the network. Each peer sends its location information on connecting to the network (Step 1). If Alice intends to download a resource, she sends a search request to this server (Step 2), which returns information about the peer (Bob) sharing the resource (Step 3). Alice then connects directly with Bob and downloads the file (Step 4). This architecture offers very good performance in terms of the response time to search requests. However, such architectures may not scale well since the central server becomes a potential bottleneck. Further, due to the existence of a central server, these schemes are not robust to attacks – an attack on the server can bring down the entire network. Napster is a very popular P2P network that uses centralized indexing. Napster collapsed due to litigations over potential copyright infringements.

2.1.2 Decentralized (Pure P2P) Networks

Decentralized networks adopt a distributed architecture for searching content shared by peers. Each peer discovers and establishes connections with a variable number of servents to exchange content as shown in Figure 2.1(b). A peer discovers new peers in the network using broadcast messages. A search request (Step 1) is broadcast from the source to the nodes directly connected to the source. Each of these nodes in turn broadcasts the request to its neighbors (Steps 2, 3), until the message reaches a peer who possesses the content or has traversed a maximum number of hops . Here, every peer is involved in handling and forwarding request messages, thus behaving both as a client and as a server. Gnutella is an example of a pure P2P network.

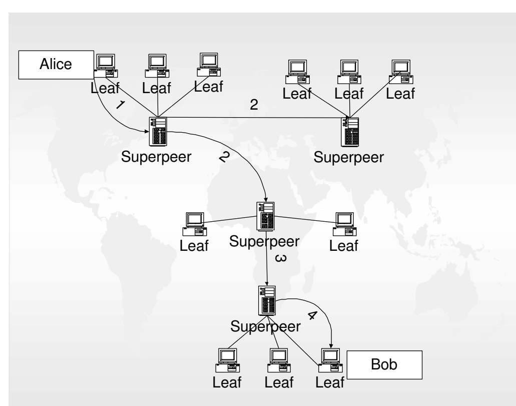
2.1.3 Hybrid Networks

Intermediate architectures have been prevalent in the recent past. In these networks, peers are classified into leaf nodes and superpeers (or supernodes), based on the computation power and amount of bandwidth of these nodes (Figure 2.1(c)). Superpeers are servents that are responsible for indexing the network content and for routing request and response messages. Leaf nodes only provide content to the network and are shielded by



(a) Centralized Network Architecture

(b) Decentralized Network Architecture



(c) Hybrid Network Architecture

Figure 2.1: Architectures in P2P Networks

the superpeers from any signaling messages, thus establishing a two-tier hierarchy among the servents. As shown in the figure, a request originating at a leaf node (or superpeer) is processed only by intermediate superpeers. The software developed by Sharman Networks – Kazaa is an instance of such networks. However, many of the Gnutella servents have adopted the hybrid model because it provides benefits from both centralized and decentralized models.

2.2 Reputation Management

There has been a lot of work in the past on reputation-based trust management for client/server based networks. An important instance of successful reputation management is the centralized online auction system eBay [22]. In this reputation system, buyers and sellers rate each other at the end of the transaction. A cumulative rating is computed for each buyer and seller based on the above individual ratings. This rating is stored on a central server managed by eBay. Hence, the above system uses a centralized approach to manage and store reputation information.

The model proposed by Yu and Singh [43] builds a social network among agents that supports participants' reputation both for expertise (providing service) and helpfulness (providing referrals). Every agent keeps a list of its neighbors (which can be changed over time) and computes the trustworthiness of other agents by updating the current values based on testimonies obtained from reliable referral chains. A bad experience with an agent results in the decrease of its rating and this information is propagated throughout the network.

The proposal for computing and using reputation for Internet ratings by Chen et al [15] differentiates the ratings by computing a reputation for each rater based on the quality and quantity of the ratings it gives. However, the method is based on the assumption that the ratings are of good quality if they are consistent with a majority of the opinions. Adversaries who submit fake or misleading feedbacks can still gain a good reputation as a rater in their method simply by submitting a large number of feedbacks and becoming the majority opinion.

Proposals in reputation-based trust management for P2P networks can be broadly classified based on centralized vs distributed storage of reputation information.

2.2.1 Centralized Approaches

TrustMe [41] is motivated by the importance of anonymity in trust based systems. A secure and anonymous underlying protocol has been proposed for trust management, which provides mutual anonymity for both the trusted host and the querying peer. A peer gives a reputation rating to another peer based on all its experiences with that peer. The trust rating of the peer is an aggregation of all ratings obtained from the rest of the peers. TrustMe provides a system similar to the secret ballot for voters. The trust rating of each peer is stored on a random peer X which replies to all queries for the trust values it possesses. A peer can anonymously issue a query and get the true value without requiring to know the identity of peer X. However, this approach depends on the availability of the peers storing the reputation information.

In the work by Minaxi Gupta et al [25], reputations are computed using either one of two schemes - debit-credit reputation computation and credit-only reputation computation. This approach computes reputation as a function of two variables - peer's behavior and its capabilities (processing power, memory, bandwidth and storage capacity). Hence, parameters such as size of shared files, bandwidth and time are used in computing the reputation. Using a centralized reputation computation agent (RCA), a public-key based mechanism has been proposed to address the problem of reliable reputations. The RCA is contacted periodically by peers to get credits for their contribution in the system based on the above schemes. Thus, the RCA maintains state information about all transactions made by all the peers in the network.

Centralized approaches are useful in consolidating reputation data at one central location. However, they may not scale well and may be susceptible to security attacks such as denial of service attacks. Thus, as part of this work, we were inclined to choose reputation systems that were decentralized in their approach.

2.2.2 Decentralized Approaches

There have been many approaches proposed to distribute and store reputation information without the use of central servers. In the following paragraphs, some of the important approaches pertinent to peer-to-peer networks have been described.

[11] focuses on management and retrieval of trust-related data by using a single

P2P distributed database which stores complaints about individuals if transactions with them were not satisfactory. When an agent p wants to evaluate trust for another agent q, it sends a query for complaint data which involves q, and decides q's trustworthiness based on the returned data, using a proposed formula. Hence, complaints are the only behavioral data being used in this work. This system does not incorporate both good and bad transactions which is essential in estimating the reputation of a peer. Further, such systems may be vulnerable to DoS attacks, since a malicious node could insert any number of complaints into the system, without being detected.

The work by Yu et al. [43] is primarily about social mechanisms for regulating users in electronic communities. This work provides a distributed agent architecture to represent and propagate the ratings that participants give to each other. When evaluating the trustworthiness of any party, a peer combines its local evidence (obtained from direct transactions) with the recommendations received from others regarding the same peer. They also propose an algorithm to compute the trust index for an agent in the range [-1, 1] that was an outcome of local evidence and testimonies. This approach focussed primarily on reputations, but did not address the issues of providing integrity and authentication of messages.

EigenRep [27] uses a distributed and secure method to assign a unique global reputation value to each peer based on the history of uploads using power iteration. This work attempts to decrease the number of downloads of inauthentic files in a peer-to-peer file sharing network. In this approach, each peer locally stores its own view of the reputation of the peers it has interacted with. The global reputation of the peer is then computed using local reputation values assigned by other peers, weighted by the global reputation of the assigning peers. This work also does not focus on describing how integrity and authentication of messages are achieved.

NICE [31] is a platform for implementing cooperative distributed applications. Applications in NICE gain access to the remote resources by bartering local resources. For each transaction, each involved user produces a signed statement (cookie) about the quality of the transaction and exchanges the statement with its peers. Hence, the locally stored reputation in NICE is an indication of the satisfaction expressed by other peers that it served. Since the peers store their own reputations, no cooperation is required from other peers for storage purposes. However, to compute reputations when needed, cooperation from the other peers in the system is mandatory, just as in other reputation systems.

PeerTrust [42] is also a feedback based trust management system for peers to quantify and compare the reputation of other peers in P2P networks. In this work, three basic trust parameters are used for evaluating the trustworthiness of peers – feedback in terms of amount of satisfaction, feedback context such as the total number of transactions and the credibility of the feedback sources. The trust for a peer in this system is also a non-decreasing scalar and is subjective. Each peer is mapped to maintain a small database that stores a portion of the global trust data. Maliciousness is countered by having multiple peers responsible for storing the same database. Voting can be used if these databases differ. Trust is computed by querying multiple databases over the network. This work uses a broadcast approach to obtain ratings for a peer, thus consuming significant bandwidth.

P2PRep [17] uses a distributed broadcast polling mechanism to search and distribute reputation information in the P2P network. Public key encryption has been used to achieve confidentiality, integrity and non-repudiation of messages. In this approach, each peer is associated with an opaque identifier. Each peer maintains a history of the transactions and its experiences with other peers in its local reputation database. The protocol comprises of five different phases. In the first phase, the initiator broadcasts its request to the network, like in a standard Gnutella interchange and receives responses from the peers, which include the identifier of the providers and a *digest* of the resource. In the resource sharing and vote polling phase, the initiator enquires the network (broadcast) about the provider from which it intends to download the resource. Upon receiving this poll message, each peer checks its experience repositories and responds with votes if it has had a prior transaction with that entity. In the third phase, the initiator performs a vote evaluation on the set of votes that it has received about each provider. It then chooses the providers with the highest reputation value as suggested by the voters. The fourth phase attempts to handle some load balancing and prevent some security attacks on the protocol. In this phase, the most reliable provider is contacted to verify the fact that it exports the resource. In the final phase, the initiator downloads the resource and based on the quality of the transaction, it updates the local repository with an entry for the set of providers from which it downloaded the resource.

The authors of P2PRep also proposed XRep [21], as an extension to P2PRep, in which two separate local repositories are maintained – one with reputation information about resources and another with reputation information about the servents. In [18], they have presented the details of the implementation of a reputation aware Gnutella servent.

RCert [34] is another reputation-based approach that uses public key encryption to provide security properties such as authentication, integrity and non-repudiation. This protocol aims to provide efficient retrieval of reputation information by storing the content locally at the owner (server-side) in the form of certificates. The protocol also ensures that these certificates are secure from being modified by the owner. This approach provides an efficient mechanism for the retrieval of reputation content. Further, all messages in RCert are unicast messages.

Thus we observe that reputation-based approaches in P2P networks require co-operation between the participating nodes to ensure distributed storage and retrieval of content. We observe that existing approaches adopt either a centralized or decentralized mechanism for storing and retrieving reputation information. These approaches use public key encryption to provide security properties such as confidentiality, authentication, integrity and non-repudiation of messages. Thus, a useful reputation based P2P system must be able to store and retrieve ratings in a decentralized manner and also provide authentication, integrity and non-repudiation of messages. Only two of the above approaches provided the desired features for the P2P reputation system – P2PRep/XRep and RCert. In Chapter 4, we compare these two approaches and justify why we chose to extend the RCert protocol. We also investigate the RCert protocol in greater depth in this chapter and propose an extension to this approach to fix certain drawbacks and vulnerabilities.

Chapter 3

Review of Gnutella Design and Implementation

In this chapter, we describe the reason for choosing Gnutella for our implementation and review the design of the Gnutella protocol. We also briefly describe the security issues in Gnutella. We have left out the details of the specification and described only sections that are relevant for this thesis. A detailed description of the protocol can be obtained in the Gnutella specification [29]. In the final section, we describe Gtk-Gnutella, a unix-based Gnutella servent.

The Gnutella protocol is an open, decentralized group membership and search protocol, primarily used for file-sharing. Gnutella has been one the most popular file sharing protocol in P2P networks. However, the original design of Gnutella did not incorporate support for security, thus making it vulnerable to many attacks. We therefore chose to study the design of Gnutella and to identify how best security can be incorporated within its existing design.

3.1 Operation

There are no central servers in the Gnutella network. In order to join the system a peer initially connects to one of several known hosts (e.g., gnutellahosts.com) that are almost always available (generally, these hosts do not provide shared files). Such entities are known as *host caches*. The host cache then forwards the IP addresses and port numbers of numerous other peers that are currently online. This connection is performed over a TCP session.

Once attached to the network (having one or more open connections with nodes already in the network), nodes send messages to interact with each other. Messages can be either a request message or a response to some previously received request message. Messages can be broadcast (i.e., sent to all nodes with which the sender has open TCP connections) or simply back-propagated (i.e., sent on a specific connection in the reverse direction on the path taken by an initial, broadcast, message). Several features of the protocol facilitate this broadcast/back-propagation mechanism. First, each message has a randomly generated unique identifier. Second, each node keeps a short memory of the recently routed messages, used to prevent re-broadcasting and implement back-propagation. Third, messages are flagged with *time-to-live* (TTL) and *hops passed* fields. These messages are described in detail in Section 3.3.

3.2 Protocol Description

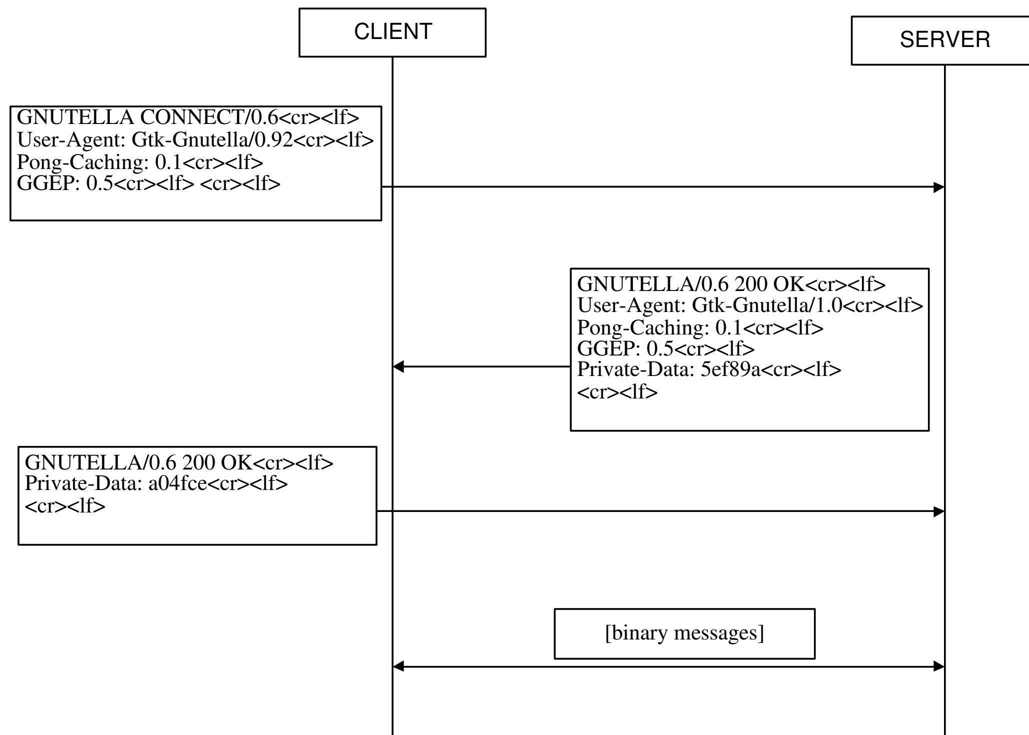
A Gnutella servent connects itself to the network by establishing a connection with another servent connected to a peer-group. Nodes forward all incoming messages to all members of their peer-group.

Each Gnutella message contains a globally unique identifier (GUID) in its header. When a node receives a message, it stores this message ID in a table before sending it on to its peer-group. If the message ID is already in the table, it discards the message since it has already been processed. After receiving the responses, the peer selects the resources it desires to download. Resources are downloaded using the standard HTTP protocol.

A sample interaction of the initial handshake between a client and a server servent has been shown in Figure 3.1:

Headers follow the standards described in RFC822 [19] and RFC2616 [23]. Each

Figure 3.1: Client-Server handshake in Gnutella



header is made up of a field name, followed by a colon, and then the value. Each line ends with the sequence $(< cr >< lf >)$, and the end of the headers is marked by a single line. Each line normally starts a new header, unless it begins with a space or a horizontal tab, in which case it continues the preceding header line.

3.3 Protocol Messages

After being connected successfully, a peer communicates with other servents by sending and receiving Gnutella protocol messages. Each message is preceded by a message header with the byte structure given below.

3.3.1 Message Header

The message header is 23 bytes and comprises of the fields as shown in Table 3.1

Table 3.1: Gnutella Message Header

Message ID (GUID)	Payload Type	TTL (Time to live)	Hops	Payload Length
0-15	16	17	18	19-22

A brief description of the elements in the header is provided below:

Message ID: A 16-byte string (GUID) uniquely identifying the message on the network.

Payload Type: This field indicates the type of message. The following codes are currently recognized: 0x00 = Ping, 0x01 = Pong, 0x02 = Bye, 0x40 = Push, 0x80 = Query, 0x81 = Query Hit

TTL: Time To Live. This field is similar to the TTL value in the IP header. The value represents the number of times the message will be forwarded by Gnutella servents before it is removed from the network. Each servent will decrement the TTL before passing it on to another servent. When the TTL reaches 0, the message will no longer be forwarded and is dropped.

Hops: This field indicates the number of times the message has been forwarded. As a message is passed from servent to servent, the TTL and Hops fields of the header must satisfy the following condition: TTL(0) = TTL(i) + Hops(i) where TTL(i) and Hops(i) are the values of the TTL and Hops fields of the message, and TTL(0) is maximum number of hops a message will travel (usually 7).

Payload Length: The length of the message (in bytes) immediately following the header is stored in this field. The next message header is located exactly this number of bytes from the end of this header i.e. there are no gaps or pad bytes in the Gnutella data stream.

3.3.2 Message Types

Immediately following the message header is the payload consisting of one of the messages shown in Table 3.2.

Table 3.2: Gnutella Message Types

Descriptor	Description
Ping	Used to actively discover hosts on the network. A servent receiving a Ping descriptor is expected to respond with one or more Pong descriptors.
Pong	The response to a Ping. Includes the address of a connected Gnutella servent and information regarding the amount of data it is making available to the network.
Query	The primary mechanism for searching the distributed network. A servent receiving a Query descriptor will respond with a QueryHit if a match is found against its local data set.
QueryHit	The response to a Query. This descriptor provides the recipient with enough information to acquire the data matching the corresponding Query.
Push	A mechanism that allows a firewalled servent to contribute file-based data to the network.

Ping and Pong Messages

Ping messages are broadcast into the network to discover new peers and statistics about the resources that they possess. A peer receiving a ping message responds with a pong message that is routed back to the originator along the path traversed by the ping message. To conserve bandwidth, the protocol uses pong caching. When a peer receives a ping message, it responds with a pong message with all the entries in its pong cache.

Query Message

A query message contains the search criteria specified by the user. As shown in Table 3.3, the query message comprises of the search string provided by the user. It also comprises of an optional extensions block, which will be described in section 3.4. A servent decrements the TTL value by 1 before forwarding the query message to its neighbors. If, after decrementing the header's TTL field, the TTL field is found to be zero, the message is dropped.

A servent receiving a message with the same payload message and Message ID as

one it has received before, discards the message since it has already processed the message.

Table 3.3: Query Message Format

Field	Min Speed	Search Criteria	NUL(0x00)	Extensions block (optional)
Byte offset 0...1	2...N	N+1	N+1...L-1	

Query-Hit Message

The QueryHit message is sent only in response to an incoming Query message. A servent should reply to a Query with a QueryHit message if it contains data that strictly meets the query search criteria. A QueryHit message should be initially generated with hops=0 and the TTL field equal to the number of hops traversed by the corresponding Query message. The Descriptor Id field in the descriptor header of the QueryHit should contain the same value as that of the associated Query message. This allows a servent to identify the QueryHit message associated with Query messages it generated.

The format has been shown in Table 3.4. The result set is a list of results with each entry having the elements showed in Table 3.5.

Table 3.4: Query Hit Message Format

Field	No of Hits	Port	IP Addr	Speed	Result Set	Optional QHD Data	Servent ID
Byte offset	0	1...2	3...6	7...10	11...10+N	11+N...L-17	L-16...L-1

Table 3.5: Result Set Structure

Field	File Index	File Size	File Name	NUL(0x0)	Optional Result Data	NUL(0x0)
Byte offset	0...3	4...7	8...7+K	8+K	9+K...R-2	R-1

Push Message

A servent may send a Push descriptor if it receives a QueryHit descriptor from a servent that does not support incoming connections. This might occur when the servent sending the QueryHit descriptor is behind a firewall. When a servent receives a Push descriptor, it may act upon the push request if and only if the Servent Identifier field contains the value of its servent identifier. The Descriptor Id field in the Descriptor Header of the Push descriptor should not contain the same value as that of the associated QueryHit descriptor, but should contain a new value.

Table 3.6: Push Message Format

Field	Servent Identifier	File Index	IP Address	Port	Optional Push Data
Byte offset	0...15	16...19	20...23	24...25	26...L-1

3.4 GGEP Extensions

The Gnutella Generic Extension Protocol (GGEP) allows arbitrary extensions in Gnutella messages. A GGEP block is a framework for other extensions. These extensions are embedded in the trailer of Gnutella query and query-hit messages. The two peers negotiate the version of GGEP being used during the initial handshake shown in Figure 3.1.

A GGEP block always starts with a magic byte used to help distinguish GGEP extensions from legacy data which may exist. It must be set to the value *0xC3*. The magic byte is followed by any number of extensions. They are processed in the order in which they appear. Table 3.7 shows the elements of the GGEP extension.

Table 3.7: GGEP Extension Format

Field	Flags	ID	Data Length	Extension Data
Length	1	1...15	1...3	x

Based on which bit in the flag field is set, this field conveys information about the encoding and compression used in the GGEP extension. It also provides information about the number of bytes used to represent the ID. More details about each individual field

can be obtained from [29]. In our implementation, all protocol messages are embedded as GGEP extensions in the query and query-hit messages, and will be discussed in detail in Chapter 4.

3.5 Gtk-Gnutella

In this section, we mention the existence of different implementations of Gnutella and describe why we have used Gtk-Gnutella for our implementation. We highlight the salient features supported by Gtk-Gnutella and briefly describe the design.

There are many different implementations of Gnutella that are currently available. They are collectively called Gnutelliums. Limewire [7] is the most popular open-source java-based Gnutella servent. Bearshare [1] and Gnucleus [3] are the predominant servents on Windows. Some other implementations are Morpheus, Mutella, Phex, etc.

Gtk-Gnutella (Gtk-G) [24] is an open-source GUI based Gnutella servent. It is a fully featured Gnutella servent designed to share any type of file the user wishes to share. It has been developed and tested under Linux using C. Gtk-gnutella is open-source and is released under the GNU Public License (GPL). In this work, we understand the design of Gtk-Gnutella and extend the code to incorporate the reputation protocol that is described in Chapter 4.

3.5.1 Features

This section summarizes the salient features that have been implemented in Gtk-Gnutella. We provide only a brief description of the features here. Detailed information about the features are described in the Gnutella specification [29].

- *GWebCache*: The goal of the *Gnutella Web Caching System* (GWebCache) is to eliminate the initial connection point problem of a fully decentralized network. The cache is a program placed on any web server that stores IP addresses of hosts in the Gnutella network and URLs of other caches. Gnutella clients randomly connect to a cache in their list. They send and receive IP addresses and URLs from the cache. With the randomized connection it is to be assured that all caches eventually learn

about each other, and that all caches have relatively fresh hosts and URLs. The concept is independent of Gnutella clients joining and leaving the nework.

- *GNet compression*: Gtk-G implements compressed gnutella net connections to conserve bandwidth in the network. Compression is meant to be done on a per-connection basis. The *deflate* scheme is handled by the www.zlib.org library (the deflate/inflate routines). This implies that there will be a compression dictionary and history per connection on both ends, thus achieving a good compression as compared to compressing each message individually. The peers exchange this information in the *Accept-Encoding* and *Content-Encoding* headers in the initial handshake described in Figure 3.1

- *Ultrapeers*: Gtk-G provides a hierarchical Gnutella network by categorizing nodes on the network as leaf nodes and supernodes or ultrapeers, based on the bandwidth capacity and computational power of the peer. A leaf node keeps only one open connection to its neighboring supernode. A supernode acts as a proxy to the Gnutella network for the leaf nodes connected to it. This makes the network scalable by reducing the number of nodes on the network involved in routing and message handling, as well as reducing the actual traffic between them.

- *PARQ*: Passive/Active Remote Queueing (PARQ) allows a provider of a resource to enqueue requests from initiators, when the server has no upload slots available. The queueing technique has been built considering fairness and other such factors.

- *QRP*: QRP (Query Routing Protocol) has been implemented in Gtk-G for routing messages efficiently by avoiding broadcast queries on the network. In this protocol, hosts create query route tables by hashing file keywords and regularly exchange them with their neighbors. Thus, a request is forwarded only if the route table received on a connection matched the keywords in the request string.

- *HASH*: Hash/URN Gnutella Extensions (HASH) allow files to be identified and located by Uniform Resource Names (URNs) – reliable, persistent, location-independent names, such as those provided by secure hash values. This technique greatly enhances end-user experience by combining display of query results which represent the same file in spite of having different filenames. This enables parallel downloading from multiple sources (*swarming*) thus providing faster downloads.

- *PFTP*: Partial File Transfer Protocol is a protocol for sharing partial files on the Gnutella network. A partial file is a file that a host has only downloaded parts of. Partial file sharing allows files to spread faster over the Gnutella network.

The graphical user interface for Gtk-gnutella has been developed using GTK+ - The GIMP ToolKit, a multi-platform toolbar for creating GUIs.

3.5.2 Design

Gtk-Gnutella has been implemented utilizing the functionality of events and callbacks provided by GTK. Every action, such as receipt of a network message, a timeout or a button click by a user is associated with an event. Each event has callback APIs associated with them, which are invoked when the event gets triggered. We provide further details about the design of Gtk-Gnutella in Section 5.1.

We chose to build our reputation management system as an extension to Gtk-Gnutella considering its simplistic design, user-friendly GUI, the development environment used and the open-source nature of the project. Further, the GUI is also useful in displaying many advanced level, protocol specific statistics that would be useful to analyze the implementation. The details of our system that have been incorporated into Gtk-Gnutella have been provided in Chapter 5.

Chapter 4

Review and Analysis of RCert Reputation Management Protocol

The RCert protocol has been designed to provide a mechanism to determine and establish trust between nodes in a P2P network. This chapter discusses the goals of the protocol and describes the approach in detail. We extend this protocol by adding messages to address a few shortcomings of RCert. We then perform a security analysis on this protocol and present the details. Finally, we compare RCert with P2PRep, another reputation-based protocol in P2P networks and describe the rationale for extending the RCert protocol as the foundation for building our reputation management system.

4.1 Introduction

The RCert protocol was designed with the following properties in mind:

- *Distributed storage of reputation information*: This protocol does not require any centralized infrastructure. It uses a decentralized approach to retrieve and store reputation information.
- *Efficient retrieval of reputation information*: In some of the existing approaches [17],[43], a broadcast approach was used to retrieve reputation information. RCert

adopts a different approach in which each peer stores its own reputation content locally. Thus, this approach ensures that the time for retrieval of the reputation content is a constant factor.

- *Integrity and non-repudiation*: RCert uses public/private keys and digital signatures to ensure the integrity and non-repudiation of messages sent on the network. Further, it does not require any PKI infrastructure for generating these keys.
- *Efficient use of network bandwidth*: Since this protocol stores the reputation information locally, the number of messages exchanged between the peers is significantly reduced. Further, all RCert messages are unicast, thus ensuring efficient utilization of network bandwidth.
- *Global rating for each entity*: The RCert protocol uses a simple technique to compute the global rating information for each peer by considering the average of all the ratings in its history.

Each peer is identified by an opaque identifier called GUID (Globally Unique Identifier). The GUID is sent in all the messages to identify the originator of the message, as described in Section 3.2. Just as in the case of most reputation-based P2P systems, the reputation information about a peer is associated with its GUID. Each new peer starts with a default rating and accumulates new rating values as it participates in the network. If a peer wishes to build its reputation over a period of time, it needs to maintain a persistent identifier in the network. When a peer assumes a new GUID, it loses all information about the past reputation and is treated as a new member in the network.

4.2 Overview

RCert allows the initiating servent to verify and evaluate the ratings of providers before downloading a resource from them. It allows the provider to store the history of its transactions locally and makes it available to the network when requested. The protocol provides a mechanism by which the initiator is able to rate the provider at the end of the transaction and provide a rating certificate for this transaction. The initiator is able to compute the global rating for a provider dynamically (as an average of the providers' past ratings) and choose the best set of providers to download from.

This protocol uses two important entities for storing reputation information - rating certificates and recommendations.

4.2.1 Rating Certificates (RCert)

Rating certificates store the ratings received from other peers at the end of a transaction. RCerts consist of a header, a list of RCertUnits and the digital signature as shown in Table 4.1. Each RCert unit essentially contains the rating provided by the previous recommenders and the identity information about the recommender who provided this rating. These certificates are stored locally at the provider and are sent by the provider along with the query-hit message. The provider signs the RCerts and stores the signature in the *Signature* field before sending them into the network.

Table 4.1: Rating Certificates

Header	RCert $Unit_1$	RCert $Unit_2$	...	RCert $Unit_n$	Signature

The RCert header contains information about the provider and its local certificates, and is shown in Table 4.2. It contains information about the identity of the owner – the GUID and the public key. This information is used during the authentication of the RCert message. The *RCert Num* and *Previous RCert Num* fields in the header are used to provide information about the history of certificates stored. In due course of time, the provider may accumulate many certificates and it may not be efficient to send all the certificates, when requested. Hence, using these fields, the provider may send only the most recent certificates, thus conserving bandwidth. It does this by setting the *RCert Num* to the most recent certificate list and by setting the *Previous RCert Num* to the value referring to the past certificates. If the provider is able to send all the certificates in the message, it sets the *Previous RCert Num* field to 0.

Table 4.2: RCert Header Format

Owner ID	Public Key	RCert Num	Previous RCert Num

As mentioned, each RCert Unit contains information about the rating and the

provider of the rating, as shown in Table 4.3. A new RCert is appended to the list of RCerts at the end of each transaction with this peer. The *rater ID*, *IP address* and *port number* fields provide information about the identity of the recommender. The *rating* contains information about the recommenders' transaction experience with the provider. The *counter* is the sequence number for this certificate. Every new RCert has the *counter* value incremented by one. The *timestamp* field contains the value of the timestamp issued by the provider. Each RCert unit is individually signed by the recommender of that certificate.

Table 4.3: RCert Unit

Rater ID	IP Address	Port Number	Rating	Counter	Timestamp	Signature

4.2.2 Recommendations

Each peer stores a list of recommendations that contains information about the rating that it had provided to the servents from which it downloaded resources in the past (Table 4.4). It provides information about the identity of the peer for whom the recommendation was provided (*GUID*), the timestamp that was provided by that servent and status information to identify the validity of the certificate (valid or revoked). It provides information about the identity of the revoked peer (*IP address* and *port number*) if the entry was revoked. We describe the use of this information when we describe the protocol in detail in Section 4.3.

Table 4.4: Recommendations

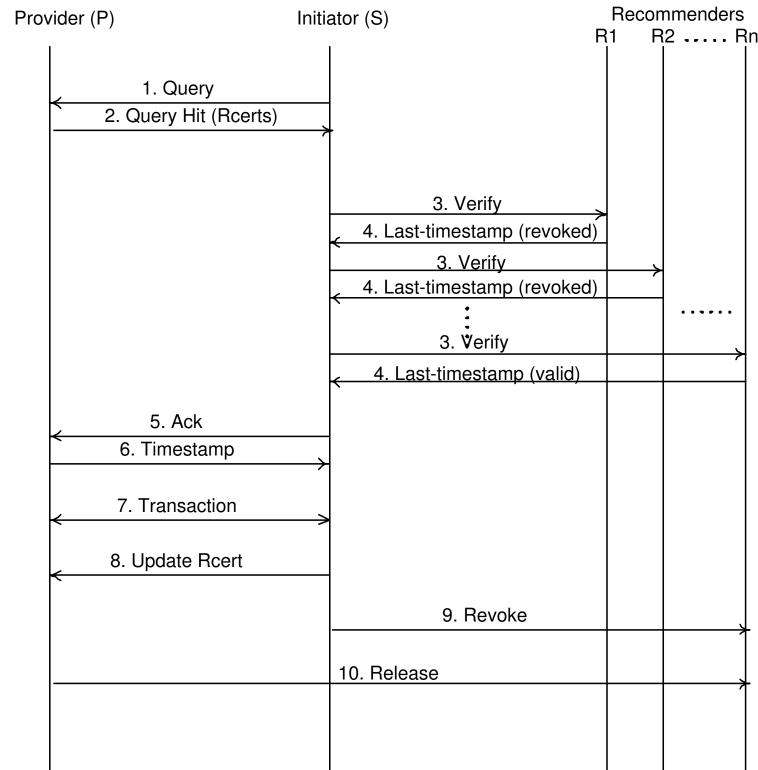
GUID	Timestamp	IP Address	Port Number	Status

4.3 Protocol description

Each transaction of the RCert protocol involves three types of entities in the network – an *initiator*, a set of *providers* and a set of *recommenders*. The *initiator* is the

entity searching for content in the P2P network. Any peer who currently shares the resource that was requested by the initiator is defined as a *provider* of that content. A peer who has had a transaction with some such provider in the past and currently possesses a rating for that provider is termed as a *recommender*. The exchange of messages between these entities has been described below. For the sake of simplicity of this discussion, Figure 4.1 shows only one provider (P) and n recommenders.

Figure 4.1: RCert Protocol Message Flow



The initiator (S) who wishes to search for a resource broadcasts a *Query* message into the network(1). Any providers (P) who are currently sharing the resource respond to this request message by sending the *QueryHit* message(2). The above steps are the same as the initial steps in the Gnutella protocol. However, in addition, the providers send their RCerts, that are locally available at the provider, in the trailer of the *QueryHit* message. The initiator verifies the signature in the response message and drops the packet if there is a mismatch.

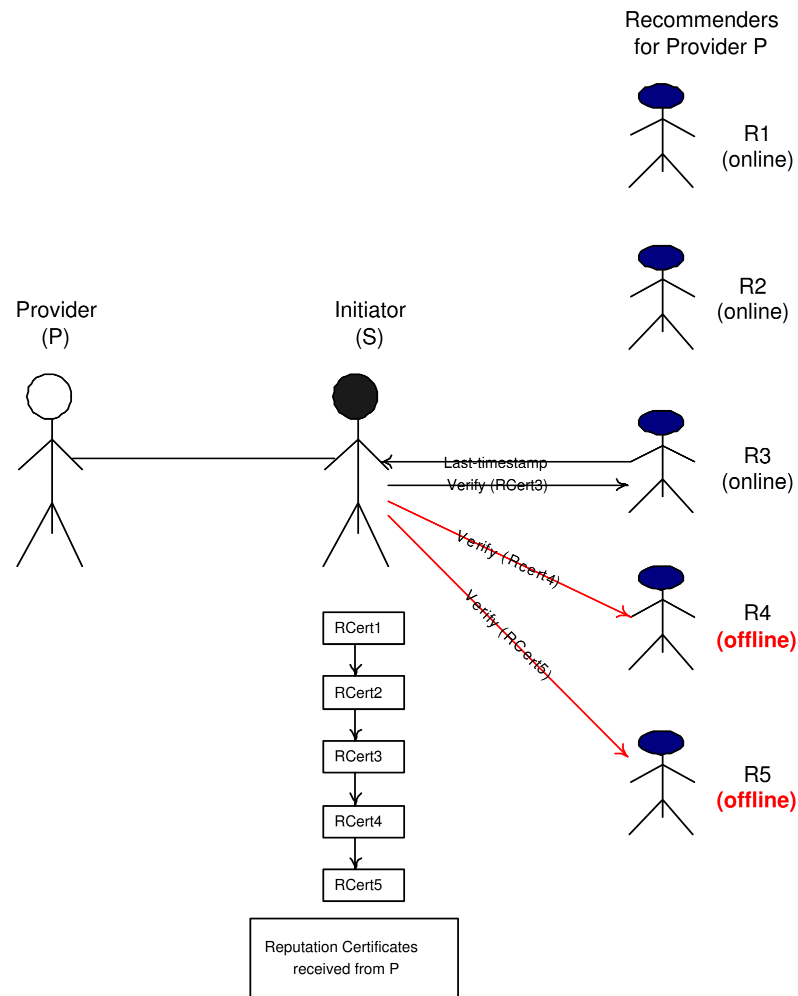
This is followed by a series of reputation messages exchanged between the three

entities. S computes the global rating for each provider from the RCerts that it received and selects the providers who have a rating higher than a predefined threshold value. The initiator needs to verify the validity of the RCerts of each of these providers. This is done by checking the last RCert unit in the received certificate list by sending a *verify* message to the recommender R1(3). The *verify* message contains the certificate that is being verified. R1 verifies the certificate and responds with a *last-timestamp* message(4). The *last-timestamp* message contains information about the timestamp that was provided by the provider, the status of the timestamp(*valid* or *revoked*) and the identity of the peer who authorized the revoke. If the recommender R1 was offline, then, the initiator repeats steps 3 and 4 by contacting the previous raters until there is one available as depicted in Figure 4.2. In this figure, we see that the most recent recommender R5 is offline and hence, the initiator S is not able to receive a *last-timestamp* response from it. Hence, it contacts R4, which is also offline. It then contacts R3 and receives a *last-timestamp* message. S verifies that the status in this message is revoked and the IP address and port number of the revoked peer matches that of the succeeding recommender in the certificate list that it originally received from the provider (i.e with the IP address and port number of R4). If verified, S computes the global rating for P using the certificate ratings upto R3's certificate (since it was not able to verify certificates $RCert_4$ and $RCert_5$. The receipt of the *last-timestamp* message terminates the process of verifying the providers' certificates.

After this step, the initiator sends a message to those providers for which it was able to successfully verify the rating certificates. This *ack* message indicates the initiator's interest in proceeding with the transaction (5). The *ack* is used as an evidence of transaction request by P. P then responds with the *timestamp* message (6). This message provides the time value on the provider's machine and the transaction counter. The initiator uses this time information in the future steps of the protocol. The transaction counter indicates the number of transactions the provider has experienced in the past. Thus, the counter reflects the latest information about the transactions of the provider.

The peers then begin the transaction, using the HTTP protocol, just as in Gnutella (7). At the end of the transaction, the initiator is provided the option of rating the service that was provided by P. The peers may consider multiple factors such as quality of content, bandwidth provided, etc, while rating the transaction. For this purpose, the initiator prepares a new RCert unit with its rating for the provider and sends it in the *update RCert* message (8). It also includes the time information that was provided by P in the *timestamp*

Figure 4.2: Protocol behavior with offline recommenders



message. On receiving this message, P verifies the value of the timestamp information in the new certificate. If the value is successfully verified, it appends the new rating to its list of certificates. At the same time, the initiator stores the last-timestamp (recommendation information) and makes it available to others, when needed. If it was not the first rater, it contacts the previous recommender to revoke the last-timestamp information on that peer (9).

The recommender R_n verifies the *revoke* message received from S and if valid, revokes the timestamp information stored locally by creating a status *revoke* and adding the identity of S who revoked the certificate. Finally, the provider notifies the previous rater that it can remove its last-timestamp certificate using the *release* message(10).

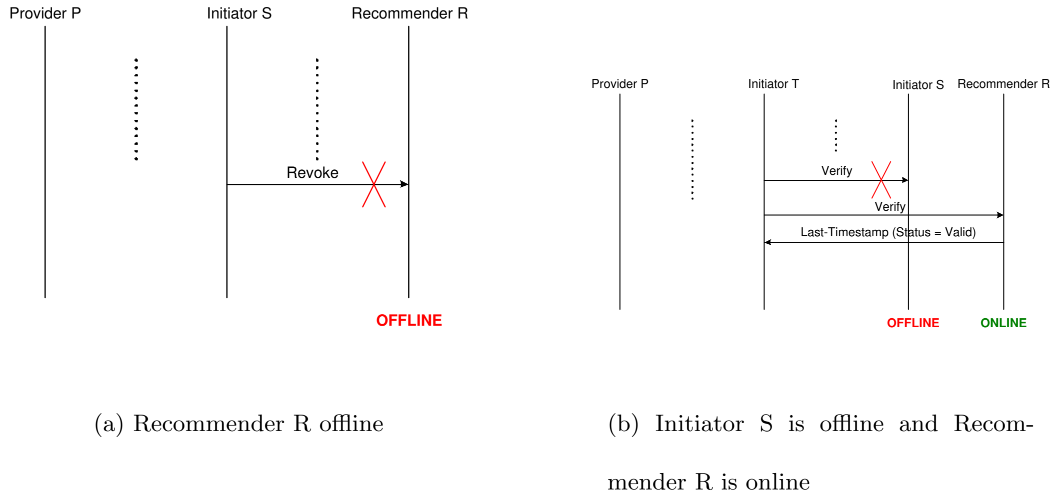
4.3.1 Discussion

The RCert protocol provides for secure local storage of rating certificates at the provider. We also call this server-side storage, since the provider acts as the server in a transaction. Hence, reputation information about a provider is always available, as long as the provider is online. The protocol allows the initiator to verify these rating certificates by contacting the recommenders in temporal order, starting from the most recent recommender. Based on the ratings available in these certificates, the initiator is able to choose the best set of providers from whom he/she wants to download the resource. The initiator would require to contact past recommenders only in instances where the most recent recommender is unavailable. Further, in these situations, the past recommenders are able to verify the rating certificates for the initiator. The *last-timestamp* message serves to verify rating certificates. After verifying the validity of the certificates of all chosen providers, the initiator sends *Acks* to these providers indicating an interest in proceeding with the transaction. The *Ack* message serves as evidence of a transaction for the provider for future references. If the initiator refuses to give a rating, then the provider can use the *Ack* message as evidence of the transaction. This message can be provided to the previous recommender to request arbitration, as required.

The protocol also provides a mechanism to rate the provider at the end of the transaction. The servent is allowed to rate the transaction with a binary rating (0 or 1) based on different factors such as quality of transaction, authenticity of shared content, etc. The protocol does not define what factors are required to be considered by the servents. Further, the global rating for a servent is computed as the average of its past ratings. While this protocol allows for such a simple mechanism to provide ratings, any other approach may be used in place of the above method. For example, the approach described in [44] uses a weighted average approach for computing the global rating. [43] applies Dempster-Shafer calculus to calculate the ratings for agents.

The protocol also provides a mechanism by which the initiator is able to revoke the timestamp from the most-recent recommender. Thus, at the end of a transaction, the initiator has a new valid recommendation in its recommendation list, the provider has a new certificate appended to its list of rating certificates and the most recent recommender has the corresponding recommendation revoked in its recommendation list.

Revocations for offline recommenders: In this paragraph, we describe the



(a) Recommender R offline

(b) Initiator S is offline and Recommender R is online

Figure 4.3: Protocol behavior when most-recent recommender is offline

behavior of the protocol when the most-recent recommender is offline when the *revoke* message is sent. We observe that this property does not affect the normal behavior of the protocol because the provider stores the most recent certificates and the last certificate in its certificate chain points to the appropriate recommender. We consider the following case where there is a peer S (initiator), peer P (provider) and peer R1 (most-recent recommender) who is offline, as depicted in Figure 4.3(a).

S completes a transaction with P and sends a new rating to P. Now, when it sends the *revoke* message to R1, R1 does not acknowledge it since it is offline. At some later instance, S goes offline and R1 logs back into the network as shown in Figure 4.3(b). A new initiator T requests certificates from P. P sends its list of certificates. T contacts S (most-recent recommender), but does not receive any response since S is offline. Hence, T contacts the previous recommender R1 who responds with the *last-timestamp* message with a valid timestamp. At this point, since T has verified the signature on the list of RCerts that it received from P, it is aware that it possesses the most recent recommendations and that R1 has provided an incorrect timestamp. Hence, it can send a message asking R1 to update its information by sending the information about S (the peer who was supposed to have revoked the certificate).

Interleaved/Concurrent sessions: The authors of the RCert protocol do not

specify the behavior of the protocol in the presence of interleaved transactions. In this paragraph, we suggest changes in the protocol that are able to handle multiple sessions started simultaneously with the same provider. In any distributed protocol that involves multiple writers, the resource being written must be locked such that we can ensure consistency in the interpretation of the resource. In the context of the RCert protocol, the shared resource being written is the rating certificate chain at the provider. In the RCert protocol, let us assume multiple initiators desire to begin simultaneous sessions with provider P and rate the provider at the end of their respective transactions. As a constraint, the protocol can allow all transactions to proceed serially - the provider can serve only one initiator at a time. However, in P2P networks such a constraint is undesirable since it could lead to denial of service attacks. We suggest a simple extension to RCert that allows simultaneous sessions with the provider. At the end of the transaction, the initiator can signal the provider (write request) that it is interested in providing a new rating, before sending the *update rcert* message to the provider. At this point, the provider grabs a lock for this session, if available, and sends its latest certificate chain. Following this step, the provider queues write requests received from other initiators, since it is still in the process of appending a new rating certificate to its certificate chain. After receiving the new rating from the initiator, P will release the resource and respond to any other initiators waiting in the queue to send a new rating certificate.

4.4 Protocol enhancements

In this section, we describe the shortcomings of the RCert protocol and propose a few improvements to address these problems. We highlight the two new messages that have been added into the protocol. We also describe the security extensions that have been incorporated into RCert to address certain vulnerabilities.

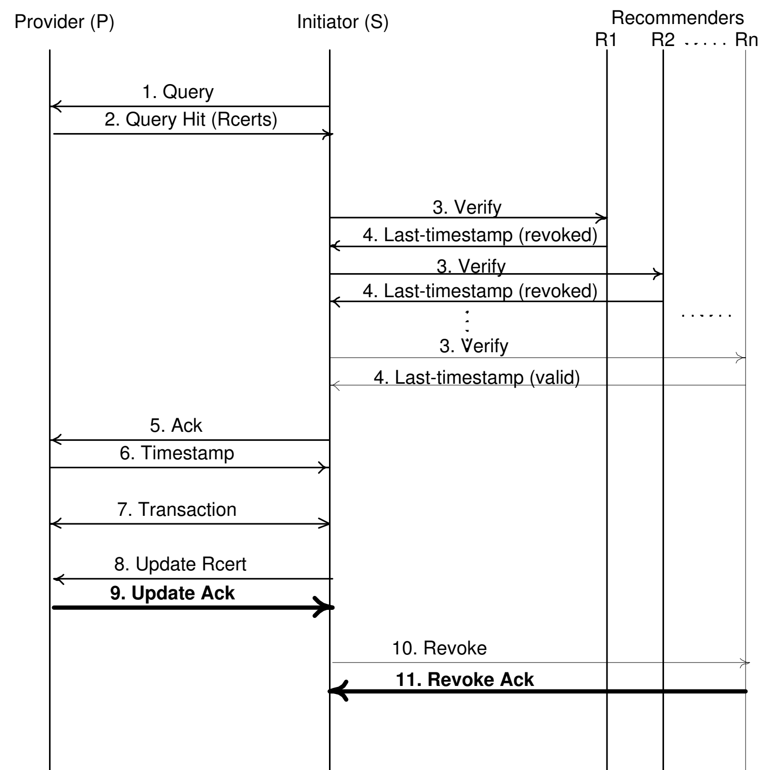
4.4.1 Message Extensions

In the RCert protocol, the initiator does not expect any acknowledgement for the *Update RCert* message sent to the provider. It sends the *Revoke* message to the recommender assuming that the provider received the *Update RCert* message and that it appended

the new rating to its certificate list. However, if the provider was offline when the initiator sent this message, the protocol may enter an unstable state, where the most recent recommender has a different notion about the status of certificates compared with the notion at the provider.

To prevent such a situation, we introduce two additional messages into the protocol as shown in bold in Figure 4.4. The *Update RCert* and *Revoke* messages are acknowledged using the *Update Ack* and *Revoke Ack* messages. In the current protocol, the initiator does not expect an acknowledgement to the *Update* message that it sends to the provider. Thus, there is no mechanism by which the initiator can be ensured that the provider has received the message. Hence, in Figure 4.4, after sending an *Update RCert* message(8), the initiator waits for an *Update Ack* message(9). Only after receiving this message, S sends the *Revoke* message(10). Again, it waits to receive an acknowledgement to the revoke message (11). The initiator re-sends these messages, if it does not receive an ack from the servents. Further, we have also removed the *Release* message, since this message was no longer required as part of the protocol.

Figure 4.4: RCert Protocol Message Flow (Modified)



4.4.2 Security Extensions

The RCert protocol lacks a mechanism of providing message authentication. The receiver of a message will not be able to verify the initiator of that message, since there is no information being provided in the messages. Lack of message authentication could lead to ID spoofing or ID stealth attacks. We propose a mechanism to authenticate RCert protocol messages. Further, since each of the RCert units are signed individually, the provider would be able to remove certificates selectively from its history. The onion signature scheme proposed in this section provides a solution against this problem.

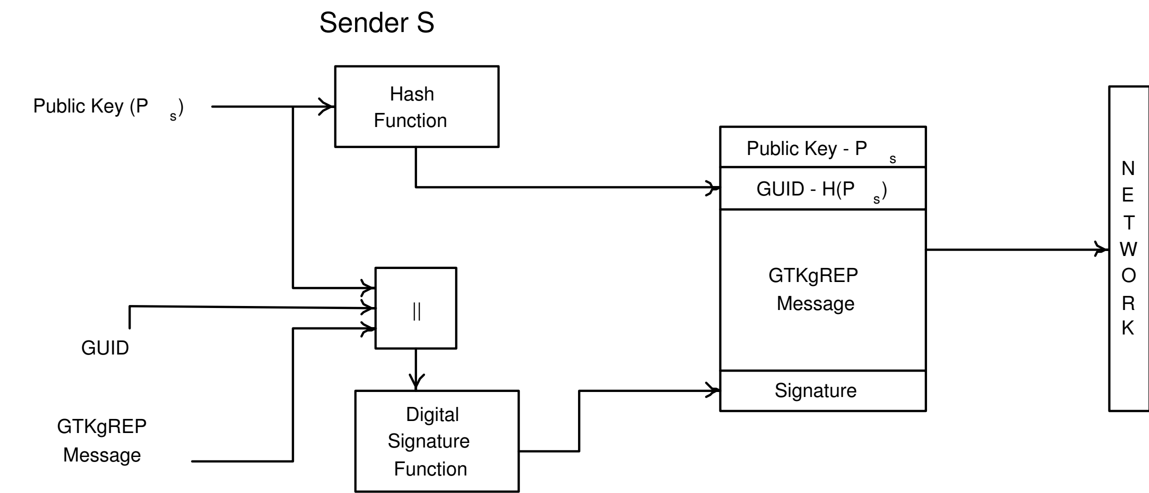
Authentication

We propose a mechanism by which the peers are able to verify the authentication of the sender's messages. The approach we have used is to include essential information about the sender's identity within each protocol message. We then ensure that there is a mechanism by which we can detect tampering of these messages in the network using public key encryption.

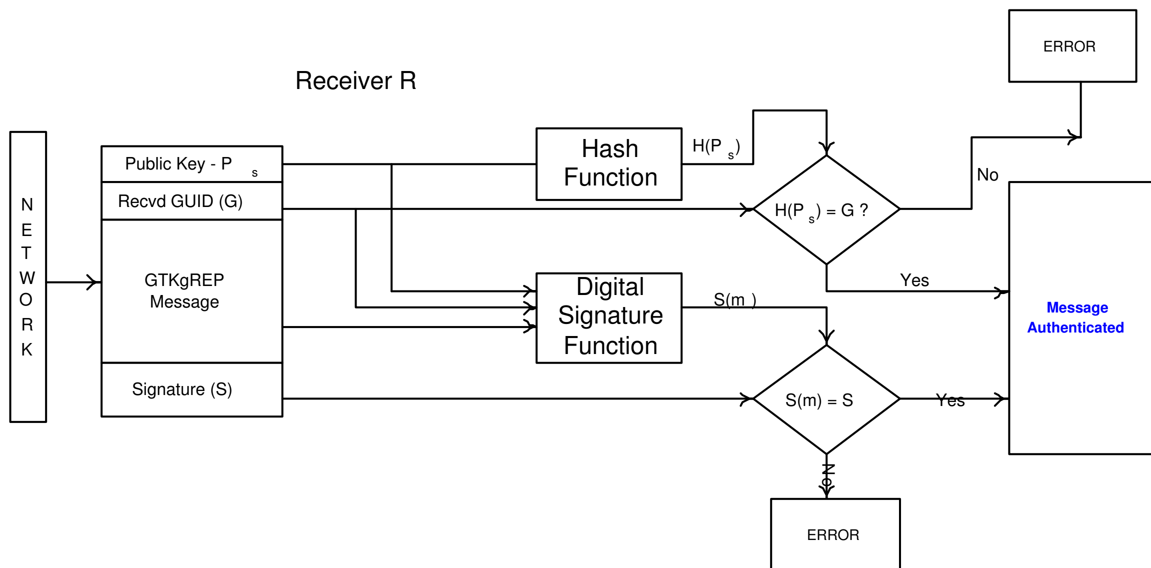
The opaque identifier of each peer (GUID) is computed as the secure hash (md5 digest) of the public key of that peer. Each protocol message includes the GUID and public key of the peer (collectively termed as *identity* of the peer) who sent the message. This information along with the message payload is signed digitally using the private key of the peer before being sent into the network. On the receipt of a message, the receiver first verifies the digital signature on the message. If verified, it implies that this message was received from the network untampered.Now the recipient verifies the GUID present in the message. This ensures that the message is verified for authenticity before being processed at the receivers end. The process of message authentication is depicted in Figure 4.5. In Figure 4.5(a), we show the process by which the sender (S) prepares and dispatches a protocol message and Figure 4.5(b) depicts how this message is authenticated at the receiver (R).

Onion Signatures

As discussed in Section 4.3, the RCert protocol provides a mechanism by which rating certificates are stored locally by the owner. Each rating certificate is signed individually by the corresponding recommender as was described in Table 4.1 and Table 4.3. The digital signatures were intended to protect the certificates from being modified by the owner. Here, we discuss a situation where a malicious provider is able to tamper with its



(a) Message creation process at the sender's side



(b) Message authentication at the receiver's side

Figure 4.5: Message Authentication in Extended RCert

certificates and thus increase its cumulative rating. We then describe the concept of onion signatures that provides a solution against this vulnerability.

A malicious provider may be able to selectively remove certificates which have received negative ratings from its history as depicted in Figure 4.6. The figure shows a provider with seven transactions in its history, which includes two bad transactions that it made in the past. We observe from this figure that the provider is able to remove certificates 2 and 3 from its history and create a new chain of certificates excluding these bad transactions. The same argument holds for the fact that the provider will also be able to add spurious certificates into its history thus increasing its global rating. This is possible because all certificates are not verified by the peer receiving these certificates (only the most recent certificates would be verified). More importantly, this drawback is due to the fact that each recommender computes a signature only on its own certificate while sending a new rating certificate using the *Update RCert* message.

Thus, in order to protect the integrity of the local certificate database, an *onion layered signature scheme* can be used by the recommenders while sending the *Update RCert* Message. In this method, the recommender incorporates its signature over all the previous certificates that it received from the provider, in addition to its new rating certificate. This can be visualized as having layered signatures, where each layer contains the certificate and signature provided by the previous recommenders – the innermost layer comprises of the information from the first recommender and the outermost layer comprises of the information from the most recent recommender.

This method is illustrated in Figure 4.7. Let us assume that R1, R2 and R3 are the only recommenders, P is the provider and S is the initiator. Let us assume that R1, R2 and R3 undergo transactions with P in that order. In this method, when R1 sends the *Update RCert* message, it signs the only certificate RCert1 with its digital signature S1, since it is the first recommender. When R2 completes its transaction, it provides a signature encapsulating R1's signature. Thus RCert1, S1 and RCert2 become the payload to compute the digital signature S2 on R2. Similarly, R3 would provide a signature S3 encapsulating R1's and R2's payload and signatures. Thus, if a provider modifies any part of its local certificate database, the digital signature will fail to match and thus its malicious behavior can be detected.

Figure 4.6: Selective Certificate Deletions

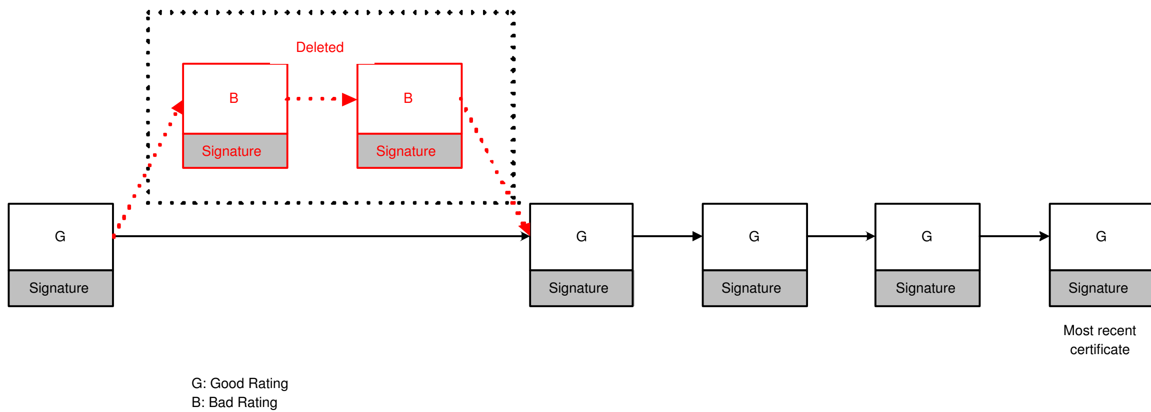
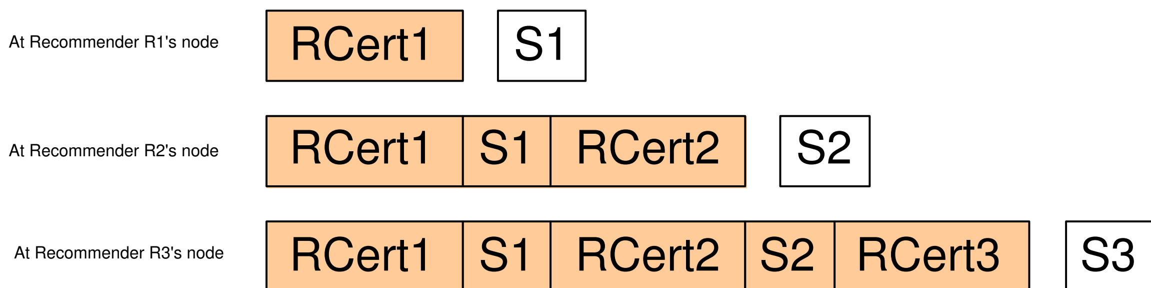


Figure 4.7: Onion Signatures in RCert



4.5 Security analysis

In this section, we perform a security analysis on the RCert protocol after incorporating the extensions described in Section 4.4. As discussed in Section 4.3, this protocol involves three entities – the initiator who requests for resources, the provider who shares the above resource (or multiple providers who share the resource) and the recommender who has a rating for the provider (or multiple recommenders with ratings for each provider). We consider the behavior of the protocol when one or more of these entities can be malicious peers.

4.5.1 Integrity and Non-repudiation

Public key encryption is being used to maintain the integrity and non-repudiation of all RCert messages. The content of all the messages is being digitally signed using the private key of the peer. The public key and GUID are being sent in messages to verify the integrity of these messages. This approach prevents any outsider from tampering with the messages in the network.

4.5.2 Authentication of messages

The extension that was proposed in section 4.4.2 provides for the authentication of messages exchanged by the protocol. The GUID is computed using a one-way hash function as the secure hash of the public key of the peer. We know that one of the properties of one-way hash functions is *weak collision resistence* – i.e. given the hash value H(m) and the message m, it is computationally infeasible to find another message m' such that Hm') = Hm). Thus, given the public key and its hash value, it is computationally infeasible to find another public key with the same hash value. This property in conjunction with digital signatures (message integrity) safeguards the protocol against ID stealth or ID spoofing attacks.

4.5.3 Validity of RCerts

The last-timestamp message used in this protocol provides information about the validity of the certificates. The initiator contacts the previous recommender to verify the validity of the certificate. If the previous recommender responds with a last-timestamp message that has not been revoked, then, the certificate is up-to-date. Otherwise, the certificate received from the provider is not the latest information and hence, this provider can be avoided.

4.5.4 Misbehaving entities

Here, we discuss some scenarios assuming one or more of the entities can be malicious.

- *Misbehaving Providers*: The provider may not wish to divulge previous recommendations, if it has received bad ratings in the past. The essential motivation for such an act could be the fact that the provider has accumulated a rating lower than the threshold initial value it started with. Hence, to avoid such circumstances, the minimum rating that a peer can receive has been ensured to be greater than or equal to the threshold initial value.

  Two collaborative peers may conduct multiple transactions with each other and rate themselves at the end of each transaction, thus increasing their global rating value. To prevent this, only the latest rating (determined by the received timestamp) is used for calculating the cumulative rating from *each* peer. If a rating already exists for the peer, it is overwritten with the new rating. This ensures that the global rating is calculated as a factor of many independent transactions undergone with different entities, thus decreasing the effect of such collusion.

  Providers may spurious add rating certificates or delete certain certificates in its history (for which it has received bad ratings). The property of onion signatures that was suggested as an extension to RCert detects such misbehavior. Such misbehavior is detected because the digital signature on the last certificate will not match the computed signature.

- *Misbehaving Initiators*: At the end of the transaction, the initiator is expected to provide a rating of the provider about the quality of the transaction. If the initiator refuses to give a rating, then the provider can use the *ack* message in the protocol as a proof of the transaction. This message can be provided to the previous recommender to request arbitration. Although there is no mechanism defined in the protocol, it can be extended to provide the above functionality.

  In addition, the initiator may successfully *revoke* the rating certificate from the most-recent recommender without sending the *update rcert* to the provider. This could lead to an unstable state regarding the transaction. In order to prevent such misbehavior, the initiator is required to send additional information in the *revoke* using which the most-recommender is able to verify that the provider has received the *update rcert* message from this initiator. The initiator sends a signed rating-timestamp as part of the *revoke* message. This rating-timestamp is signed by the provider and appended in the *update ack* message. The rating-timestamp will include a new time value and

the identity of the rater (i.e. identity of the initiator) and is digitally signed by the provider. When the most-recent recommender receives the *revoke* message, it verifies the signature on the rating-timestamp to ensure that the initiator has sent a new rating to the provider before sending the *revoke* message.

- *Malicious Recommenders*: A malicious node may send a *revoke* message attempting to revoke a legitimate certificate stored on a recommender. The RCert protocol prevents such misbehavior. The recommender receiving this message first verifies if the timestamp is more recent than the one that it currently possesses for the provider. It also verifies whether the timestamp was indeed signed by the provider and revokes the timestamp only if the two conditions are satisfied. Otherwise, the recommender simply discards the message.

  The most recent recommender for a provider may falsely respond with a *last-timestamp* message by setting the status as revoked. This act is easily detected by the initiator receiving this *timestamp* message. The fact that this recommender was the most recent recommender on the provider's list of recommendation certificates implies that a valid *last-timestamp* message was expected from this peer. Thus a recommender will not be able to falsely respond with the status of the certificate as revoked.

4.5.5 Collusions

In non-cooperative networks such as peer-to-peer networks, a set of peers may collaborate to achieve some of their goals. For example, the provider and the most-recent recommender may collude to drop new rating information and revoke message received at the two peers respectively. Such a collusion allows the provider to drop all negative ratings that it has received for its bad transactions. The RCert protocol does not detect such misbehavior in the network. In order to detect collusions in the P2P network, certain peers in the networks can behave as watchdogs[33] to the network, whose responsibility is to detect protocol misbehavior in the network. Such peers can store the identities of misbehaving nodes and provide this information to the network, when requested. Another mechanism that can be adopted is to allow the recommenders to store the ratings at their end in case it had given a bad rating for a transaction. Hence, in this extension, in addition to verifying the providers'

certificates, the new initiator also queries the network to be informed of any negative ratings associated with a provider. Thus, the global rating would be a function of the ratings in the provider's certificates and the negative ratings received from other peers. Although this extension makes the protocol use a broadcast approach, the protocol still uses the bandwidth conservatively since the number of negative ratings are much fewer than the number of positive ratings in the network.

4.6 Extended RCert vs P2PRep/XRep

We evaluated two protocols before commencing on the implementation of a reputation system for P2P Networks - P2PRep/XRep and RCert. P2PRep was briefly discussed in Section 2.2.2. Both protocols were specific to P2P networks and had different approaches in distributing and storing the reputation information. This section highlights the security features provided by P2PRep and compares them with RCert (with the proposed extensions). It also compares the two approaches in terms of performance such as bandwidth consumed, delay, etc.

4.6.1 Security Analysis of P2PRep/XRep

Public key encryption has been used to provide integrity and confidentiality of messages. P2PRep uses hashes and signatures to detect tampered votes. A pair of keys are generated on the fly for each poll message that is being sent into the network by the initiator. When sending the poll request, the initiator includes the set of $provider_{ids}$ whose reputation is being enquired and a public key PK_{poll} generated on the fly for the poll request. The recommenders send their responses to the poll by encrypting them with this public key. Hence, P2PRep also addresses confidentiality of the votes sent by the recommenders. The advantage of this method is that the votes and the recommender's identity remain concealed. To prevent a provider from spoofing identities, there is a challenge-response phase being used just before the commencement of the download. In this mechanism, when the initiator (S) is ready to download a resource from the provider (P), S challenges P to assess whether it corresponds to the declared $servent_{id}$. P will then need to respond with a message containing its public key and the challenge signed with its private key. If the signature is

verified successfully at the initiator, then, the initiator proceeds with the download.

P2PRep uses two different variations of its protocol – basic polling protocol and enhanced polling protocol. In the former, the recommenders do not reveal their identities, whereas, in the latter they reveal their identities. While the former protocol addresses anonymity to some extent, the latter provides an opportunity for the initiator to choose votes based on the credibility of the recommenders. To prevent recommenders from sending fake poll reply messages, the protocol uses techniques based on IP clustering.

4.6.2 Comparison of the two approaches

In this section, we compare the two approaches RCert and P2PRep/XRep in terms of security and performance.

Security: Both approaches use public key cryptography to ensure integrity, authentication and non-repudiation. Both approaches use digital signatures to verify the integrity of messages. P2PRep/XRep provides a higher degree of security by incorporating confidentiality and anonymity of the participating peers. While this provides secrecy of communication, it causes significant overhead at each peer, since they would be involved in encryption and decryption for every P2PRep message processed. Further, we felt that secrecy of communication between the peers is not one of the essential goals for our reputation system.

Performance: There are different factors that we considered to measure the performance of the two approaches –

- *Bandwidth Cost*: There is one poll message being broadcast into the network for each servent or set of servents per search message in P2PRep. Broadcast causes significant bandwidth overheads to the Gnutella network. In RCert, all messages involving communication between the initiator, the provider and the recommender are unicast.

- *Computational Cost*: The P2PRep approach creates a public/private key pair for each poll message being sent into the network. Key generation is a very costly operation and hence, this approach requires significantly higher computational costs. In RCert, there is only one public/private key pair being generated for each peer to serve as its

identity. It uses the same key pair for authentication and providing integrity of all RCert protocol messages.

- *Recommenders' Availability*: P2PRep uses a client-side storage approach for storing reputation information – the reputation content is stored at the rating peer (recommender) and not at the target peer (provider). Thus, this protocol depends significantly on the availability of the recommenders, since they are required to respond to the initiator with their opinions. On the contrary, RCert provides for server-side storage of rating certificates, i.e. reputation information is stored locally at the provider. Thus, in this approach, reputation information about the provider is always available as long as the provider is online. Further, even if one of the recommenders is not available, it is still possible to contact one of the previous recommenders and successfully verify the authenticity of the certificates.

- *Reputation Repositories*: In P2PRep, when A downloads a resource from B, A (recommender) would stores the rating of B (provider) on its node. Hence, in this approach, a peer needs to provide storage to store the ratings of other peers in the network. On the contrary, RCert uses server-side repositories, since the provider stores its own rating. Hence, RCert does not expect cooperation among peers for storage of reputation information.

- *Detecting Deception*: In P2PRep, after completing a transaction, the initiator A provides a rating about the provider B and stores it locally. This information is not distributed until explicitly requested from the network. Hence, if A lies about the rating (i.e in spite of having a good transaction with B, it might give a negative rating to B), it is not detected by the protocol since this information is never broadcast or provided to B. In RCert, since the provider is able to see the rating obtained from the initiator, it is possible to detect such malicious behavior.

Thus, from the above comparison of the extended RCert protocol and P2PRep/XRep, we observed that both approaches address the required security issues of authentication, integrity and non-repudiation of messages, while providing reputations. The broadcast approach used in P2PRep/XRep would result in additional overhead in terms of the bandwidth usage. Similarly, P2PRep/XRep adds more computational overhead compared to RCert by

creating public/private key pairs for every reputation message. Thus, we choose to implement the extended RCert protocol and provide the details of the design and implementation of this protocol (GTKgREP) in Chapter 5.

Chapter 5

Design and Implementation of a Gnutella-based Reputation Management System

In this chapter we describe the details of the design and implementation of a Gnutella-based reputation management system – GTKgREP (GTK-gnutella-Reputation System). We chose to build GTKgREP as an extension to Gtk-Gnutella considering its simplistic design, user-friendly GUI, the development environment used and the open-source nature of the project. GTKgREP implements the RCert protocol with the extensions that were discussed in Section 4.4. This chapter describes the additional components added to Gtk-G, message formats, flow of events, state diagrams and test scenarios as part of the implementation. We also provide an assessment of the overheads incurred by the protocol.

For the purpose of this discussion, a transaction is defined to be an interaction between the initiator, the provider and the recommender beginning with the query message from the initiator and ending with the revoke-ack message received from the previous recommender. This includes all the intermediate messages described in Figure 4.4.

5.1 GTKgREP Subsystems

We have implemented GTKgREP as an extension to the basic implementation of Gtk-G. Figure 5.1 shows the primary subsystems involved in the implementation of GTKgREP. In this context, we use the terms *subsystem*, *component* and *manager* interchangeably. We provide this design based on an understanding of the open-source code for Gtk-G, since no design document is available. The three components in Figure 5.1 - *PKI Manager, GTKgREP Reputation System and GTKgREP Reputation Database* are the additional components that were added to Gtk-Gnutella as part of this work. In this section, we describe each of the components in further detail.

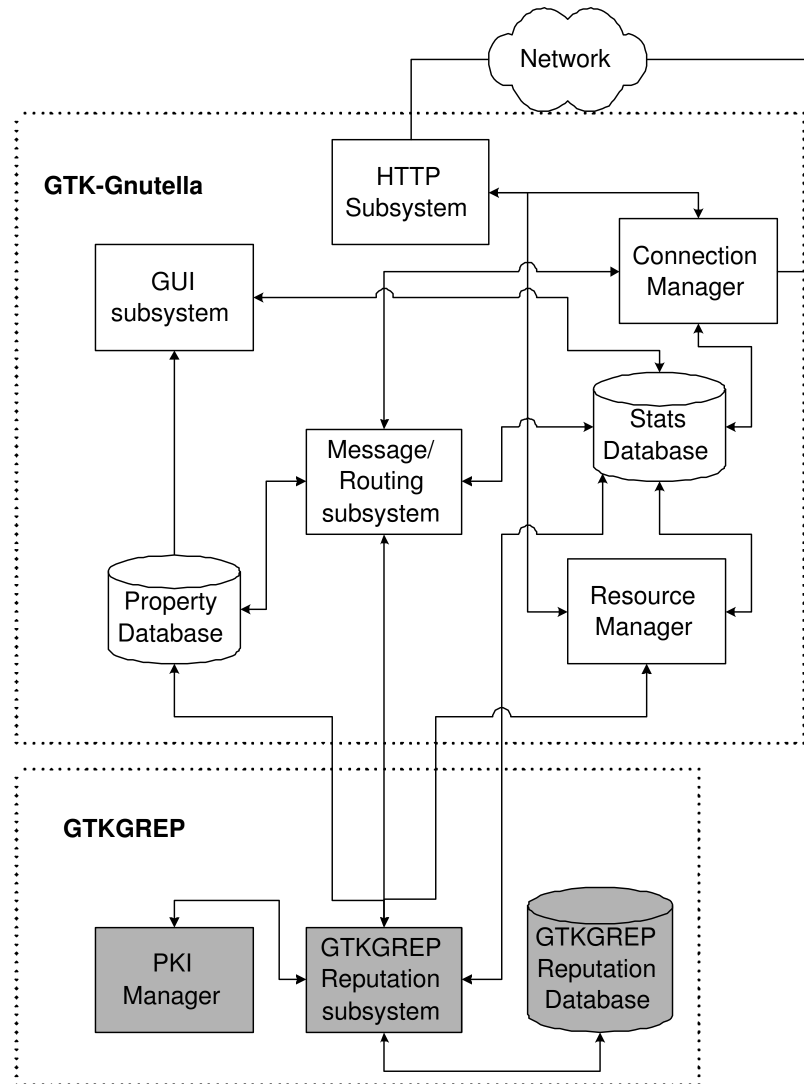
5.1.1 Message/Routing Subsystem

The message/routing subsystem is the core of Gtk-Gnutella. When a Gnutella message is received from the network, this component parses the message (including the GGEP extensions) and contacts the resource manager if the message is a query message. Further, this contacts the reputation subsystem, if a GTKgREP message was parsed. It is also involved in preparing Gnutella request/response messages based on inputs it receives from the GUI component, resource manager and connection manager. As part of message processing, it is involved in implementing GNet compression that was discussed in Section 3.5.1. This component also maintains route tables for the leaf nodes by implementing the QRP protocol, and responds to route requests on behalf of the leaf nodes, if the peer is operating as a supernode in the network.

5.1.2 Connection Manager

The connection manager stores all the information about the network connections associated with a peer. It stores information about the neighbors of a peer and is responsible for generating ping/pong messages and responding to crawler messages. It maintains information about the operating mode of the peer (leaf node or superpeer), based on the average uptime of the peer and available bandwidth (by obtaining this information from the stats component). The connection manager also implements the GWebCache which was discussed in Section 3.5.1.

Figure 5.1: GTKgREP Components



5.1.3 Resource Manager

This component is involved in maintaining information about the resources (files and directories) shared on the local node. It is also responsible for indexing all the resources and setting up query routing tables if QRP is enabled. When a search request is received from the network, the message subsystem contacts the resource manager to identify if there is a match for the search request. It receives triggered events from the GUI whenever the user adds/removes new files or directories that it intends to share.

5.1.4 GUI Subsystem

This component provides data structures for the user interface used in the implementation of Gtk-G. This component also triggers events based on inputs provided by the user and invokes appropriate actions on the GUI or in the core. This subsystem is strongly decoupled from the message/routing subsystem, which gave rise to some design issues (discussed in Section 5.2).

5.1.5 HTTP Subsystem

In Gnutella, a direct connection is established between the initiator and the provider and the data transfer is performed over HTTP. This subsystem is involved in managing the HTTP connections associated with a peer. It communicates with the connection manager and the resource manager to initiate, transfer and terminate downloads/uploads of resources.

5.1.6 Property and Stats Database

The property database manages user-configuration parameters and network-configuration parameters. User-configuration parameters include variables such as maximum number of uploads, maximum number of simultaneous downloads, etc. The network-configuration parameters include properties such as firewall configuration on the peer, address of the proxy, etc. The stats database provides different static and run-time statistics about the current session such as the number of active connections, number of ping/pong messages, number of invalid messages received, etc.

5.1.7 PKI Manager

The PKI manager provides the cryptographic infrastructure required for GTK-gREP. It provides interfaces for creating public/private key pairs associated with each peer, MD5 [36] hashes and digital signatures associated with each message sent by the peer. We use RSAREF [10], a cryptographic toolkit from RSA Laboratories to provide the above cryptographic infrastructure. RSAREF has been designed as a library (in C) whose APIs can be invoked from an application program, such as GTKgREP.

5.1.8 GTKgREP Reputation Subsystem

The reputation subsystem is the core of GTKgREP. It receives Gnutella messages from the message subsystem and parses GTKgREP messages (present as GGEP extensions) and takes appropriate actions. It also maintains state information for each transaction. This component is responsible for forming GTKgREP request/response messages and dispatching them to the network through the message subsystem. It interacts with the PKI manager to generate the public/private key pair, to form the GUID and to verify/sign all GTKgREP messages. It communicates with the reputation database to load the reputation information into the run-time environment and to save run-time configuration changes.

5.1.9 GTKgREP Reputation Database

This database stores reputation certificates received from other peers and recommendations for other peers. It also stores the history of the transactions for the owner of the database. This database is accessed by the reputation subsystem when required.

5.2 Design Issues

Among different open-source Gnutella implementations, we chose Gtk-Gnutella because of its extensive design and simplified user-interface. However, the base protocol had to be modified due to some issues in the design of Gtk-G. We discuss some significant issues in this section.

5.2.1 Stateless Protocol

The Gnutella protocol uses a request/response mechanism. Hence, it does not require much state information to be maintained at each node. However, each transaction in GTKgREP involves multiple exchanges of request/response messages, which mandates storage of state information for each transaction. Hence, the implementation was changed to store important state information on a per-transaction basis.

5.2.2 Gui-Core Dependency

In Gtk-G, the core is not independent of the GUI. All significant data structures in the core are replicated in the GUI. Hence, when events are triggered, these data structures are modified separately in the core and GUI. Recently, there has been some discussion on the Gtk-G developers forum to tightly couple these two components. However, since the source code was unavailable at the time of this work, we replicate data structures used for GTKgREP messages in the core and GUI.

5.2.3 Non-persistent GUID

Every new instance of Gtk-G created a new GUID for the session. However, in GTKgREP, a servent wishing to establish some reputation as a provider requires persistence of this opaque identifier. We made necessary modifications to the source code for this purpose. Further, to provide authentication of messages sent by this peer, the GUID is computed to be the secure hash of the public key of the peer, for which only this peer knows the corresponding private key.

5.2.4 Broadcast searches

By definition, search requests in Gnutella are broadcast to all connected nodes. However, the GTKgREP protocol needs to send unicast messages. Hence, source code in the connection subsystem and message subsystem were modified to incorporate this feature. Whenever a unicast GTKgREP message is to be sent, the GTKgREP subsystem contacts the connection manager to establish a Gnutella connection and then sends the message only to that peer.

5.2.5 Size of search messages

The implementation did not allow search requests greater than 256 bytes to be sent into the network. The Gnutella protocol draft [29] states *Servents MAY drop Query messages larger that 256 bytes, and SHOULD drop Query messages larger than 4 kB.* Since GTKgREP request messages are greater than 256 bytes, the code in the message subsystem had to be rewritten to support such messages.

5.3 Message Formats

This section details the format for all GTKgREP messages. GTKgREP request messages are sent in the Gnutella query message and response messages are sent in the Gnutella query-hit message. All GTKgREP messages are embedded in the GGEP extensions of the Gnutella message. More information about the data structures used in the implementation can be obtained from Appendix A

While *Verify, Ack, Update, Revoke* are the GTKgREP request messages, *RCert, Last-timestamp, Timestamp, Update Ack, Revoke Ack* are the GTKgREP response messages.

All request messages are obtained by modifying the Gnutella query message as shown in Table 5.1.

Table 5.1: GTKgREP Request Message Format

Field	Min Speed	Search Criteria	NUL(0x00)	Extensions block

The fields *Search Criteria and Extensions block* have been modified to have the following values:

```
Search Criteria = "GTKGREP:"<TYPE>":"
        <TYPE>      = "VERIFY" | "ACK" | "UPDATE" | "REVOKE"
Extensions Block = <EXTTYPE><LENGTH><VALUE>
        <EXTTYPE>   = Verify | Ack | Update | Revoke
        <LENGTH>    = Length of the extension
        <VALUE>     = <GUID><EXTN_ELEMENTS>
            <GUID>           = GUID of peer to whom request is to be sent
            <EXTN_ELEMENTS>  = The associated extension elements
```

All response messages only modify the filename field in the results set (Table 3.5) in the query hit message. This field is set to the string ”GTKGREP:RESULTS” to indicate that the query hit is in response to a GTKgREP request message.

Table 5.2 shows the message formats of each of the individual GTKgREP request and response messages that are embedded within GGEP extensions. It also shows the sizes of each field (in bytes).

Table 5.2: GTKgREP Message Formats

RCert (92 bytes)

Field	Rater ID	IP Address	Port	Rating	Counter	Timestamp	Signature
Length	16	4	2	1	1	4	64

RCert Header (148 bytes)

Field	Identity	Certificate Num	Prev Num	Signature
Length	80	2	2	64

Certificates (212 + 92n bytes)

Field	RCert Header	$RCert_1$	$RCert_2$	.....	$RCert_n$	Signature
Length	148	92	92	.....	92	64

Verify Message (236 bytes)

Field	Identity	Certificate	Signature
Length	80	92	64

Last timestamp (91 bytes)

Field	GUID	Timestamp	Status	IP Address (Optional)	Port (Optional)	Signature
Length	16	4	1	4	2	64

Ack (148 bytes)

Field	Identity	ACK	Signature
Length	80	4	64

Timestamp (152 bytes)

Field	Identity	Time	Transaction Counter	Signature
Length	80	4	4	64

Update RCert(236 bytes)

Field	Identity	RCert	Signature
Length	80	92	64

Update Ack(148 bytes)

Field	Identity	Ack	Signature
Length	80	4	64

Revoke Ack(148 bytes)

Field	Identity	Ack	Signature
Length	80	4	64

5.4 State Machine

This section provides the state machine for the GTKgREP implementation for the initiator and provider. We do not indicate a state machine for the recommender, since the protocol only defines request-response transactions for recommenders and does not require storing any additional state information.

Figure 5.2 shows the state machine for the two entities that have been implemented in GTKgREP. The part of the figure labelled *State Machine - Initiator* represents the states and transitions for an initiator and the part of the figure labelled *State Machine - Provider* represents the states and transitions for a provider. If there is any kind of error in any state of the protocol, such as incorrect signature, invalid GUID, etc, the protocol terminates in the error state for that transaction.

5.5 Flow of events

This section describes the interaction between the components of GTKgREP in detail. It depicts the invocations made on the three additional components (PKI Manager, GTKgREP Reputation System and GTKgREP Reputation Database) that were added as extensions to the existing code base of Gtk-Gnutella.

Figure 5.3 shows the sequence diagram for GTKgREP when the user wishes to send a search request. On starting up the client the connection manager looks up the GWebCache entries in its database and sends connection requests as part of the peer discovery process. It also exchanges ping/pong messages with the network. When the user enters a search string, the GUI makes a call onto the message/routing subsystem to initiate a search request, which in turn sends a query message into the network. After receiving replies to the query, the servent would pass them on to the User Interface, which is responsible to display the results to the user. At this point, our code intercepts the message and passes it into the GTKgREP reputation subsystem to parse the reputation extensions. After verifying the signature, GUID and the RCerts in the query response, the GTKgREP subsystem requests the connection manager to set up a connection with the recommender and prepares the *Verify* message. Again the handle is passed to the message/routing subsystem to dispatch the *Verify* message.

A similar sequence of calls follow for each of the request/response messages follow-

Figure 5.2: GTKgREP State Machine

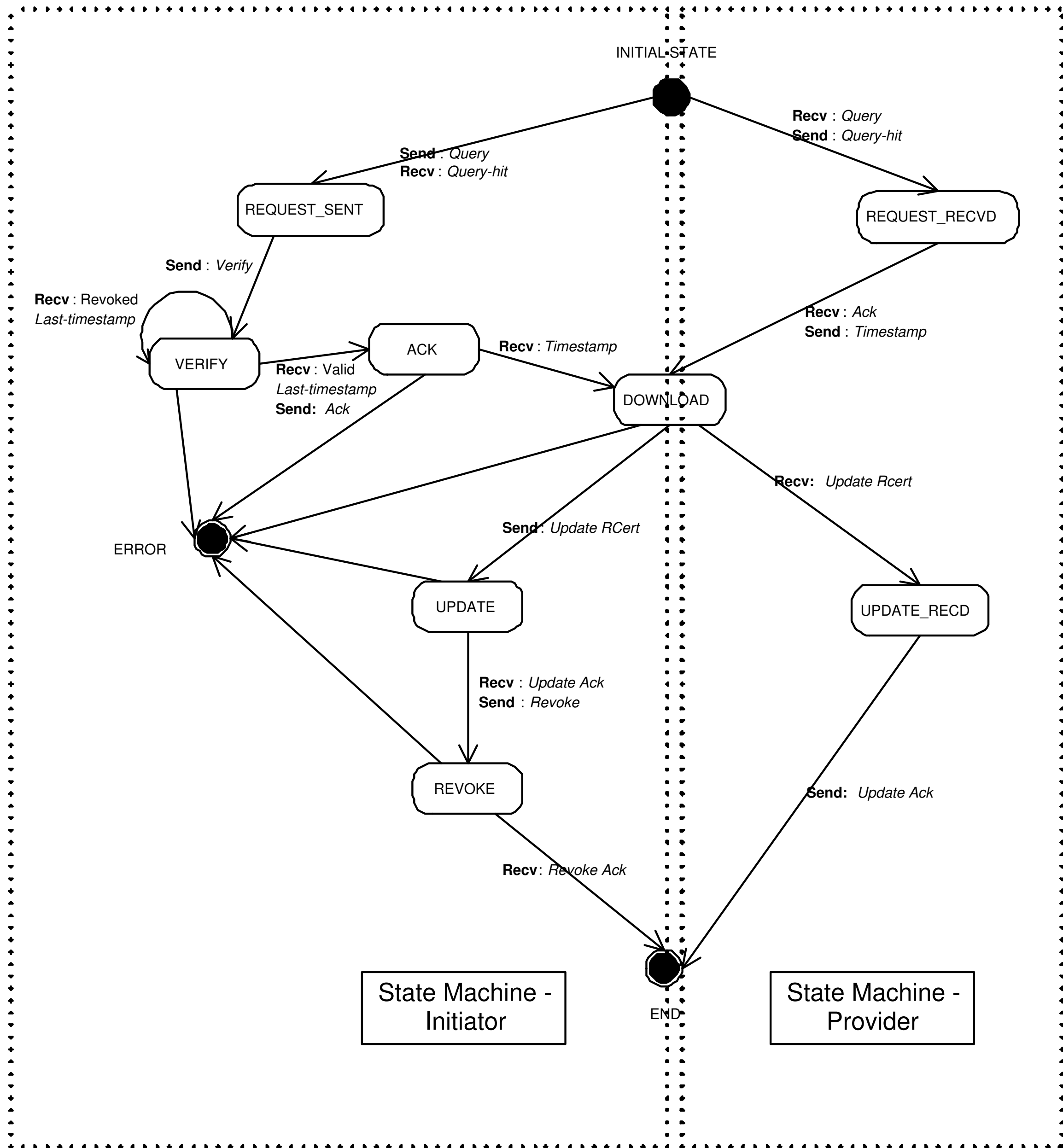
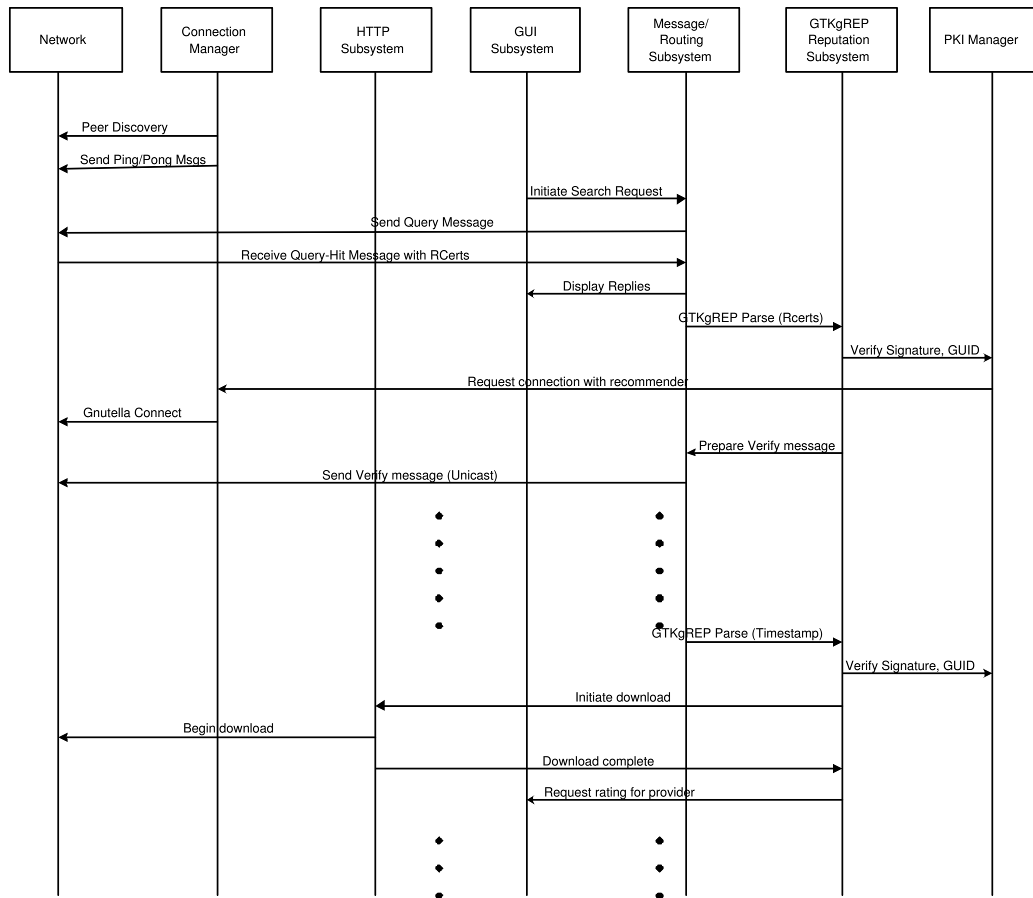


Figure 5.3: GTKgREP Sequence Diagram



ing the *Verify* message up to the beginning of the download process, described in Figure 4.4. For brevity, this information has been represented as *dots* in the sequence diagram. After receiving the *timestamp* message, the GTKgREP subsystem contacts the HTTP subsystem to initiate the download. The HTTP subsystem establishes an HTTP session with the provider and begins transfer of the file.

After completion of the download, the GTKgREP component contacts the GUI to initiate an interface for providing a rating to the provider. When the user is ready to provide the rating, the sequence of interactions between the components are similar to the one described above starting from the dispatch of the *Update RCert* message to the receipt of the *Revoke Ack* message.

5.6 Test Scenarios

This section describes some test cases that were examined to verify the implementation. These test cases were intended to verify the correctness of the protocol. We examine the protocol in cases where the participants follow the protocol as defined. We also identify a few cases of abnormal behavior among the entities and provide results from these scenarios.

We used a test bed that was set up for this purpose on the intranet. The test bed comprised of four participating nodes as shown in Figure 5.4. These nodes were randomly chosen to behave as the initiators, providers and recommenders in different network configurations.

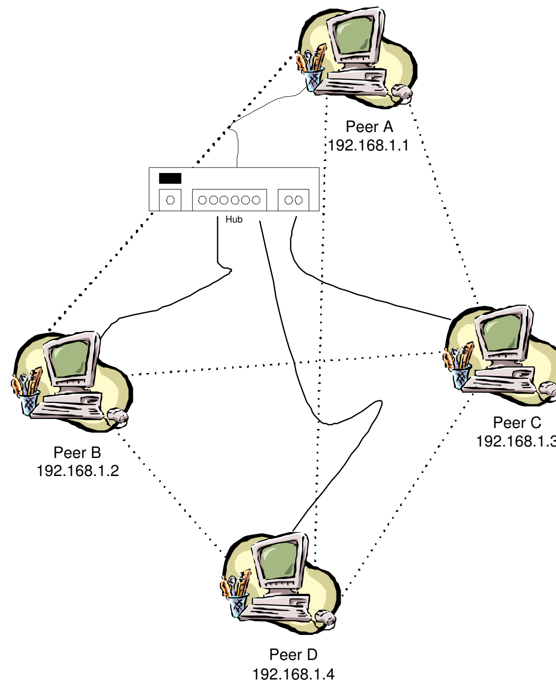
While logging into the network using Gtk-Gnutella, a peer contacts a well-known host cache to discover its neighboring peers in the network. Since the test bed was within the intranet, there was no publicly available host cache to discover its neighbors. Thus, each peer had to be manually configured to be connected to the other peers on the test bed. This provided us the flexibility to configure different network scenarios between the peers. The dotted lines in Figure 5.4 indicate the possible connections between peers within the test bed.

Bootstrap: The bootstrap involved successful creation of a public/private key pair for each peer and computation of the GUID as the secure hash of the public key of the user. This information was stored in the Gtk-Gnutella configuration file. In all future transactions, the information for the GUID and key pair was obtained from this configuration file. Each peer starts with a default rating (threshold) and creates and stores a certificate for itself on startup, since it has no transactions in its history – all peers startup as being their own recommenders. Thus, a servent is able to identify that it is communicating with a new peer by examining this certificate information.

Normal transaction: In this setup, we tested a simple transaction between an initiator and a provider, in the presence of one recommender for that provider. The initiator was able to successfully verify the signature on the rating certificates received from the provider. We also verified the authentication of messages exchanged between the initiator, provider and recommender. As part of this test case, the network was set up to have two different configurations:

- *Directly connected peers*: In this scenario, the initiating peer was assumed to be

Figure 5.4: P2P Test Bed



connected directly to the provider and recommender (neighbors). Hence, GTKgREP messages were directly sent and received by the three entities. The initiator was able to successfully download a file from the provider and provide a new rating for this transaction. It was verified that all the entities updated their reputation and recommendation databases at the end of this transaction.

- *Routed Peers*: The setup was modified to allow the initiator to be able to communicate with the provider and recommender who were multiple hops away. Hence, the GTKgREP messages were being routed by intermediate nodes to the destination entities. Just as in the previous case, we tested a successful transaction. Since this case was carried out following the previous case, the global rating for the provider was verified to incorporate the rating from the previous transaction.

Multiple Providers: The shared resources on the nodes were modified to have multiple providers sharing the same resource (file) in the network. On sending a search request for this file, the initiator received a response from multiple providers, indicating that they possessed the content. The initiator was able to see the rating for each of these

providers on the GUI and was able to select the best providers from this list. It then initiated the download from these reputed peers. At the end of the transaction, the initiator provided a rating that was sent to all the providers from whom the file was downloaded.

Multiple Recommenders: We modified the network configuration and the certificates stored, such that, the network consisted of one initiator, one provider and two recommenders for this provider. We considered two cases that tested scenarios when the most recent recommender was available and was offline.

- *Most recent recommender online*: This scenario was similar to the case in which we described a normal transaction. Since the most recent recommender was available, the initiator was able to receive a *last-timestamp* message verifying the providers' certificates. The transaction proceeded as expected and the initiator was able to download the file from the provider.

- *Most recent recommender offline*: In this test case, we were able to test the behavior of the protocol where the initiator had to contact previous recommenders to verify the providers' rating certificates. The most recent recommender was intentionally shutdown for this purpose. When the initiator did not receive a response from the most recent recommender for its *verify* message, it contacted the previous recommender (who was online). On receiving the *last-timestamp* message from the previous recommender, the initiator verified that the status in this message was revoked and that the IP address and port number corresponded with that of the most recent recommender.

Tampered Certificates: This test case was able to verify the onion signatures described in the extended RCert protocol. We tested a scenario where a provider tampered with his/her certificates. We spuriously added new certificates at the provider and deleted some old certificates that had received bad ratings. When the initiator received the rating certificates from the provider, it detected that the digital signature on these certificates were incorrect. The certificates were dropped and the transaction failed in such a case. Similarly, we tested verification of digital signatures on all other messages and dropped them if the signature in the message did not match with the computed signature.

5.7 Assessment of overheads

In this chapter, we provide an assessment of the overheads incurred on the Gnutella network, while providing the security features discussed. We provide statistics about the bandwidth consumed on the network, the delay encountered and a count of the number of public key encryptions performed by the protocol.

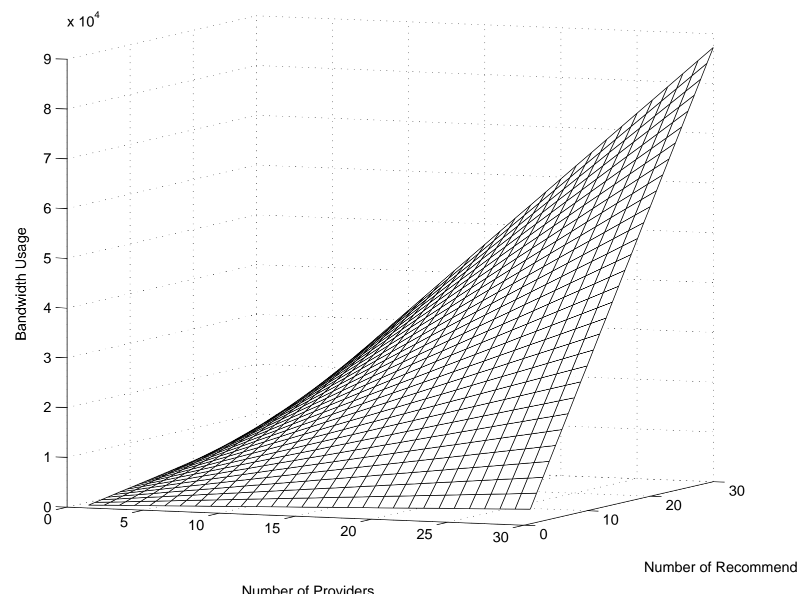
5.7.1 Bandwidth Usage

The total bandwidth consumed on the network by the exchange of RCerts when an initiator receives responses from n providers, each of which have m recommenders is given by the formula:

$bandwidth_{rcert}$ = n * (rcert_header_size + (rcert_size * m))

We plot the bandwidth consumption as a function of the number of providers and number of recommenders in Figure 5.5 for n = 30, m = 30. From this plot we observe that even in the worst case of 30 providers and 30 recommenders, the maximum bandwidth used by the RCert messages is of the order of 85Kb.

Figure 5.5: Certificate Bandwidth vs Number of providers and recommenders



The total bandwidth consumed by the protocol for a transaction would involve

bandwidth contributions from all the GTKgREP protocol messages. For the purpose of computing the total bandwidth, we assume the ideal case in which the initiator contacts only the most recent recommender and receives a successful *last-timestamp* message. The total bandwidth is given by the sum of all the message sizes as below:

$bandwidth_{total} = rcertheader_{size} + \text{history} * rcert_{size} + verify_{size} + lasttimestamp_{size} + ack_{size} + timestamp_{size} + update_{size} + updateack_{size} + revoke_{size} + revokeack_{size}$

Figure 5.6: Total Bandwidth Consumption

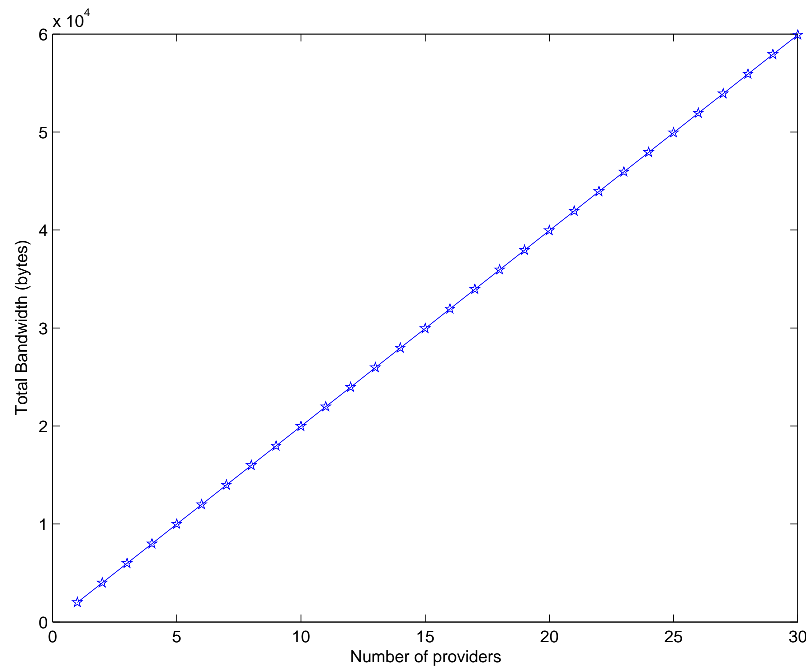


Figure 5.6 plots the total bandwidth consumed for the ideal case of contacting only the most recent recommender. In this graph, the size of the history for each provider is assumed to be fixed at five certificates. We observe the linear usage of bandwidth with the increase in the number of providers. The total bandwidth consumed can be further reduced in the presence of GNet compressions. We do not have results to indicate the percentage of bandwidth compressed using this mechanism.

5.7.2 Round Trip Delays

In this section, we summarize the delay incurred in GTKgREP. The original Gnutella protocol needed two round trips to complete a transaction. In GTKgREP, there

Table 5.3: Number of Signatures Computed/Verified

Message	RCert	Verify	Last-TS	Ack	Timestamp	Update/Ack	Revoke/Ack
Initiator	1	1	1	1	1	3	2
Provider	2	-	-	1	1	2	-
Recommender	-	1	1	-	-	-	2

Table 5.4: Total Signatures and Hashes

	Initiator	Provider	Recommender
Number of signatures computed	6	2	2
Number of signatures verified	4	4	2
Number of hashes (MD5 Digests) computed	4	3	2

are additional round trip messages being exchanged. To identify the number of round trip times spent in this protocol, we assume the ideal case in which the initiator contacts atmost one recommender and receives a successful *last-timestamp* message. We observe that the protocol would require six round trip times to complete a transaction.

5.7.3 Cryptographic Statistics

GTKgREP uses public key cryptography to sign and to verify signatures. Thus, in this section, we summarize the number of public key computations encountered in the protocol. Each peer computes a new public/private key pair when it first joins the network. This would also include situations where the peer changes its identity. Further, the peer uses one hash to compute its GUID as the hash of its public key. Thus, bootstrap involves one public/private key generation and one hash.

Subsequently, signatures are computed before sending GTKgREP messages and are verified on receiving a message. Further, hashes (MD5 digests) are computed on receiving these messages. Table 5.3 indicates the number of signatures computed or verified for each GTKgREP message by each entity.

Table 5.4 identifies the total number of signatures and message digests computed for every transaction. This table gives an indication of the total overhead incurred by the protocol in terms of encryption.

In this chapter, we have highlighted the design incorporated to build our Gnutella-based reputation management system. We have also described the implementation details

of GTKgREP. We have thus realized the practical deployment of such a system in peer-to-peer networks. We have summarized the overheads incurred by the protocol in terms of bandwidth consumed, round-trip delay and number of encryptions.

Chapter 6

Conclusion and Future Work

This work identifies and analyzes different approaches to build reputation systems in peer-to-peer networks. We identify one such reputation system - RCert and compare it with another existing approach P2PRep, in terms of benefits, performance and security.

RCert uses public key cryptography to provide message integrity. It involves a sequence of messages being exchanged between three entities - the initiator, provider and recommender to exchange and stores reputations and to rate each other about the quality of transactions. We chose to extend RCert because of its efficient use of bandwidth and computational resources. The protocol was extended to incorporate additional messages to allow peers to rate each other at a later time, thus allowing them to evaluate the quality of the transaction. We also identified the vulnerabilities of this protocol and provided solutions against the same. The protocol was extended to include message authentication and onion signatures to prevent the certificate owner from tampering with his/her certificates. We performed a security analysis on the RCert protocol and identified its benefits and features.

We have provided the design and implementation of the extended protocol - GTK-gREP using Gtk-Gnutella, a Gnutella client on Linux. As part of the design, we provide the state machine for the initiator and provider. We describe the components added into Gtk-Gnutella to incorporate the functionality and highlight the data flow between these components using sequence diagrams. We have provided results of our implementation indicating the additional overhead incurred at the cost of providing security.

6.1 Future Work

Our work can be extended to detect and prevent deception in the network. Presently, if the initiator provides a malicious rating at the end of the transaction (For eg: rates a good transaction with a zero rating), there is no mechanism to prevent or punish such misbehavior. Thus, further research could address this issue.

Another direction of research is to attempt incorporating GTKgREP into hybrid networks where leaf nodes and superpeers exist in the system. Here, only the superpeers would be involved in storing reputation messages and in responding to requests on behalf of the leaf nodes. This approach can give additional benefits in terms of performance of the network, since reputation messages would eventually be processed only by high-bandwidth superpeers.

Bibliography

[1] Bearshare - the world's best gnutella client. http://www.bearshare.com.

[2] The freenet project. http://freenet.sourceforge.net.

[3] Gnucleus website. http://www.gnucleus.com.

[4] Gnutella website. http://gnutella.wego.com.

[5] Gnutella-worm mandragore. http://www.viruslist.com/eng/viruslist.asp?id=4161.

[6] Kazaa media desktop. http://www.kazaa.com.

[7] Limewire: Running on the gnutella network. http://www.limewire.com.

[8] Morpheus website. http://www.morpheus.com.

[9] Napster website. http://www.napster.com.

[10] Rsaref - a cryptographic toolkit. http://www.spinnaker.com/crypt/rsaref/.

[11] Karl Aberer and Zoran Despotovic. Managing trust in a peer-2-peer information system. In Henrique Paques, Ling Liu, and David Grossman, editors, *Tenth International Conference on Information and Knowledge Management (CIKM01)*, pages 310–317. ACM Press, 2001. http://citeseer.nj.nec.com/aberer01managing.html.

[12] Eytan Adar and Bernardo Huberman. Free riding on gnutella. In *Technical report, Xerox PARC*, August 2000. citeseer.nj.nec.com/adar00free.html.

[13] K. Bennett and C. Grothoff. Gap - practical anonymous networking. In *Proceedings of the Privacy Enhancing Technologies Workshop (PET 2003)*, March 2003. citeseer.nj.nec.com/bennett02gap.html.

[14] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of Fifth Symposium of Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, USA, 2002.

[15] M. Chen and J. P. Singh. Computing and using reputation for internet ratings. In *Third ACM Conference on Electronic Commerce*, 2001.

[16] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: Distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009, 2001. citeseer.nj.nec.com/clarke00freenet.html.

[17] F. Cornelli, E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Choosing reputable servents in a p2p network. In *Eleventh International World Wide Web Conference*, Honolulu, Hawaii, May 2002.

[18] F. Cornelli, E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Implementing a reputation-aware gnutella servent. In *International Workshop on Peer-to-Peer Computing*, Pisa, Italy, May 2002.

[19] David H. Crocker. Standard for the format of arpa internet text messages. http://www.faqs.org/rfcs/rfc822.html.

[20] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*, october 2001.

[21] Ernesto Damiani, De Capitani di Vimercati, Stefano Paraboschi, Pierangela Samarati, and Fabio Violante. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *Nineth ACM conference on Computer and communications security*, pages 207–216. ACM Press, 2002. http://doi.acm.org/10.1145/586110.586138.

[22] Ebay. The world's online marketplace. http://www.ebay.com.

[23] R. Fielding et al. Hypertext transfer protocol – http/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[24] Yann Grossel and Raphael Manfredi. Gtk-gnutella - the graphical unix gnutella client. http://gtk-gnutella.sourceforge.net/.

[25] Minaxi Gupta, Paul Judge, and Mostafa Ammar. A reputation system for peer-to-peer networks. In *Thirteenth International ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2003. citeseer.nj.nec.com/573330.html.

[26] Sandvine Incorporated. Peer-to-peer file sharing. The impact of file sharing on service provider networks, http://downloads.lightreading.com/wplib/sandvine/P2P.pdf.

[27] Sepandar Kamvar, Mario Schlosser, and Hector Garcia-Molina. Eigenrep: Reputation management in p2p networks. In *Twelfth International World Wide Web Conference*. Stanford University, May 2003.

[28] M. Katz and C. Shapiro. Systems competition and network effects. *Journal of Economic Perspectives*, 8(2):93–115, 1994.

[29] Tor Klingberg and Raphael Manfredi. Gnutella 0.6, January 2002. http://rfc-gnutella.sourceforge.net/draft.txt.

[30] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Nineth International ACM Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, November 2000.

[31] Seungjoon Lee, Rob Sherwood, and Bobby Bhattacharjee. Cooperative peer groups in nice. In *IEEE Infocom 2003*, April 2003. citeseer.nj.nec.com/lee03cooperative.html.

[32] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *IEEE Symposium on Security and Privacy, Oakland*, May 2002. citeseer.nj.nec.com/533810.html.

[33] Sergio Marti, T. J. Giuli, Kevin Lai, and Mary Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *Mobile Computing and Networking*, 2000.

[34] Beng Chin Ooi, Chu Yee Liau, and Kian-Lee Tan. Managing trust in peer-to-peer systems using reputation-based techniques. *Fourth International Conference on Web Age Information Management (WAIM'03)*, August 2003.

[35] Paul Resnick, Ko Kuwabara, Richard Zeckhauser, and Eric Friedman. Reputation systems. *Communications of the ACM*, 43(12):45–48, 2000.

[36] R. Rivest. The md5 message-digest algorithm, April 1992. http://www.faqs.org/rfcs/rfc1321.html.

[37] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.

[38] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[39] V. Scarlata, B. Levine, and C. Shields. Responder anonymity and anonymous peer-to-peer file sharing. In *Nineth International IEEE Conference on Network Protocols (ICNP)*, 2001. citeseer.nj.nec.com/scarlata01responder.html.

[40] Vitaly Shmatikov and Carolyn Talcott. Reputation-based trust management. In *Journal of Computer Security*, 2003. citeseer.nj.nec.com/576212.html.

[41] Aameek Singh and Ling Liu. Trustme: Anonymous management of trust relationships in decentralized p2p systems. In *Third International IEEE Conference on Peer-to-Peer Computing*. Georgia Institute of Technology, September 2003.

[42] Li Xiong and Ling Liu. Building trust in decentralized peer-to-peer electronic communities. In *Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002.

[43] Bin Yu and Munindar P. Singh. A social mechanism of reputation management in electronic communities. In *Cooperative Information Agents*, pages 154–165, 2000. citeseer.nj.nec.com/yu00social.html.

[44] Giorgos Zacharia, Alexandros Moukas, and Pattie Maes. Collaborative reputation mechanisms in electronic marketplaces. In *HICSS*, 1999.

Appendix A

Data Structures in GTKgREP

```
/**************************/
/* GTKgREP data structures */
/**************************/

/* Structure: RCert
 * Data structure that maintains rating certificate information
 */
typedef struct _rcert {
    gchar rater_id[16];
    guint32 ip_addr; /* IP address of the peer who rated this (current) peer */
    gint16 port;     /* Port number of the peer who rated this peer */
    gint8 rating;    /* Numerical value of rating provided ranging from 0 to 1*/
    gint8 counter;   /* For later use */
    gint32 timestamp;/* For later use */
    guchar digital_sig[DIGITAL_SIGNATURE_LEN]; /* This is the signature of the
                                                * content of the certificate
                                                * that was signed by the
                                                * peer who provided the rating
                                                * */
```

```
} rcert;

/* Structure: Identity
 * Data structure that contains a peer's identity information
 */
typedef struct _identity {
    gchar owner_id[16]; /* GUID of the owner who sent this message */
    guchar public_key[PUBLIC_KEY_LENGTH]; /* public key of the owner who
                                             sent this message */
} st_identity;

/* Structure: RCert Header
 * Data structure that maintains header information about the list of RCerts stored
 * locally by the provider
 */
typedef struct _rcert_hdr {
    st_identity identity;
    guint16 cert_num;     /* Certificate number */
    guint16 prev_cert;    /* Previous certificates, 0 if none */
    guchar digital_sig[DIGITAL_SIGNATURE_LEN];
} rcert_hdr;

/* Structure: Global State List
 *  A new state element is created for each provider within a
 *  transaction
 */
typedef struct _global_state_list {
    gchar provider_guid[16]; /* GUID of the provider */
    gchar recommender_guid[16]; /* GUID of the recommender */
    state_table_element state; /* Current state of the transaction */
} st_global_state_element;
```

```
/* Structure: Pending Searches
 * Data structure that maintains state information about a
 * transaction
 */
typedef struct _pending_searches {
    gchar   provider_guid[16];     /* GUID of the provider */
    gint32  provider_ip;           /* IP address of the provider */
    gint16  provider_port;         /* Port number of the provider */
    gchar   recommender_guid[16];  /* GUID of the recommender */
    gint    timestamp;             /* Timestamp received from the provider */
    gint8   rating; /* This value is between 0 and 1 and is the rating that
                     *  is provided by the user at the end of the transaction
                     */
    gint    update_status; /* one of grep_state_init, grep_state_sent,
                            *        grep_state_ack */
    gint    revoke_status; /* one of grep_state_init, grep_state_sent,
                            *        grep_state_ack */
} st_pending_searches;

/* Structure: ts_element
 * This data structure stores the timestamps sent to the
 * initiator who requested for a resource. A new element of this type is
 * created every time a new ACK message is received and a new TIMESTAMP element
 * is being created
 * The timestamp is then matched to an incoming UPDATE RCERT message to ensure
 * that we have received the update from the right peer
 * */
typedef struct _ts_element {
    gint timestamp;              /* Timestamp sent to the initiator */
    gchar remote_guid[16];       /* GUID of the initiator */
} st_ts_element;
```

```
/*************************/
/* PKI Data Structures     */
/*************************/

/* RSA public and private key.
 */
typedef struct {
  unsigned int bits;                          /* length in bits of modulus */
  unsigned char modulus[MAX_RSA_MODULUS_LEN];                  /* modulus */
  unsigned char exponent[MAX_RSA_MODULUS_LEN];          /* public exponent */
} R_RSA_PUBLIC_KEY;

typedef struct {
  unsigned int bits;                          /* length in bits of modulus */
  unsigned char modulus[MAX_RSA_MODULUS_LEN];                  /* modulus */
  unsigned char publicExponent[MAX_RSA_MODULUS_LEN];    /* public exponent */
  unsigned char exponent[MAX_RSA_MODULUS_LEN];         /* private exponent */
  unsigned char prime[2][MAX_RSA_PRIME_LEN];              /* prime factors */
  unsigned char primeExponent[2][MAX_RSA_PRIME_LEN];  /* exponents for CRT */
  unsigned char coefficient[MAX_RSA_PRIME_LEN];         /* CRT coefficient */
} R_RSA_PRIVATE_KEY;
```