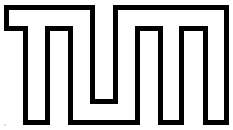


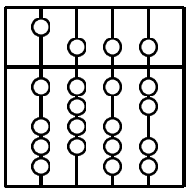
INSTITUT FÜR INFORMATIK

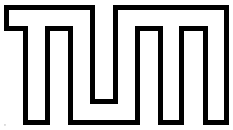
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Managing Trust in a Distributed Network

Bearbeiter:	Benedikt Elser
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Helmut Reiser Latifa Boursas Pierangela Samarati





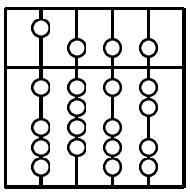
INSTITUT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Managing Trust in a Distributed Network

Bearbeiter:	Benedikt Elser
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Helmut Reiser Latifa Boursas Pierangela Samarati
Abgabetermin:	14. März 2006



Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 14. März 2006

. . . . . .

(Unterschrift des Kandidaten)

This thesis serves as an introduction to the topic of trust and appropaches to its management in a distributed network. It will show how to use reputation as a way to minimize the possibility of frauds when encountering unknown clients. For this purpose various techniques, that are currently available, will be analyzed. In the course of this work, a distributed hash table, called the Kademlia network, will be adapted to feature a reputation based trust system. A distributed hash table enables a mapping between keys and buckets across a whole network, without the need for a central index system. The reputation information will be aggregated in this fashion by collecting recommondations of peers, about others, at distributed, but well known places. This information will be guarded using cryptographic hashes, to prevent any tampering. The system will therefore be totally decentralized and enable secure reputation sharing.

Contents

List of Figures

List of Tables

1 Introduction

Over the last several years, the Internet as a business platform has gained momentum. Systems like amazon and eBay prove that effective business can take place on a “virtual” basis. But this is not only the case in business. Even communication is increasingly shifting towards technologies provided by the global network. Email, Instant Messaging and online shopping are becoming more and more part of every day life.

Besides the undoubted benefits, this rapid growth served also as a catalyst for the evolution of criminality. Not a week passes without media reports of new waves of mail viri and people fooled by a hostile eBay vendor. A new trend in criminality are phishing attacks that try to retrieve sensitive banking data from customers of online banking. Judging from the rising amount of these frauds, it is evident, that abusing others on the Internet is a lucrative task and works fairly well. Various factors influence this. This work will try to limit this problem, by analyzing the employed forms of interaction.

Big differences in the interaction can be found when comparing a confrontation with an unknown peer in real life and on a virtual basis, like the Internet. In familiar forms of interactions peers are able to collect “meta information” from each other. This includes the appearance and the facial expression of the other party. Facing a vendor competence or interaction with the desired product enables a customer, to develop confidence that his decision is correct.

Internet style interaction differs greatly from that. It is much easier for a peer to control which information can be collected. An example illustrates this: Creating a webshop which looks trustworthy for users is much cheaper than this is in real life. The physical unavailability makes judging harder for us.

This comparison demonstrates, that normal style interaction enables us to develop a rather common element of social interaction: “Trust”. While every form of human interaction is based on this property, cyberspace as new media lacks of mechanisms to judge the trustworthiness of peers.

1.1 Motivation

No matter what kind of transaction we face on the Internet, it will mostly involve totally unknown peers communicating with each other. Whenever a peer is confronted with the decision of transacting with another peer it inevitably comes to a trust decision, identical to normal social interaction. Regarding a `http` transaction of a website, one server provides information to n clients. Chapter 2.7 will however show, that regarding *Peer to Peer* communication, the situation changes drastically, as n servers, provide information to n clients, which boosts the number of trust decisions, while also stressing the number of unknown clients. While it is common when facing a stranger in real life, to collect additional information and base a judgement on that, the Internet lacks these possibilities. Therefore a trust decision is not really possible, as we can base our judgement on only a few factors and assumptions.

This problem is normally limited, as a number n of clients will try to access services of m provides, with $m << n$. Recent years however introduced a new trend of distributed services, the *Peer to Peer* systems. In those systems every client not only uses, but also provides services to others, with $m == n$. The benefit is that the number of service providers increases. On the other hand the number of interactions with unknown clients increases drastically, which increases the probability of facing a hostile peer. The complete scenario can be found in chapter 2.7. As currently no “ready to use” solutions exist, this work will try support current research.

1.2 Conceptual Formulation

The overall goal of this work is to enable trust based decisions on the Internet, trying to overcome the previous lack. To reach this goal a couple of steps have to be taken:

- Analyze the concept of trust in sociology. To develop a model in computer science, the surroundings of "real world" trust decisions need to be analyzed briefly. This is done to ensure, that the developed model represents an accurate abstraction of reality.

- Identify the components of trust in computer science. Current research on trust has developed a set of formalism and split into a variety of research areas. An overview of this complex topic is given, as an introduction to the complex topic in the informatics domain.

- Give an overview of the current state of research. As there have already been different approaches to the topic in computer science, an introduction into the current research in the surroundings of trust will be given. What different approaches to the topic of trust have already been taken and what can provide valuable input for this research.

- Present an abstract model. After receiving input from different trust research topics, a model that enables trust based decisions on the Internet will be proposed. With this model, an attempt will be made to contribute to the current state of research in the surroundings of both P2P systems and trust systems.

- Develop an implementation. The proposed solution will be included into a major P2P application.

1.3 Proceeding

To reach the previously defined goals, the second chapter of this thesis will introduce the formal surroundings of trust, starting with the definition of the word trust, that will be used throughout the course of this work. Subsequently a generic example developed by the theoretical field of computer science will lead to the requirements of trust in computer science. Finally the scenario of this work will be introduced.

Chapter 3 will give an introduction on the current state of research. It will illustrate past models that were developed to develop trust. During the course of this chapter, these solutions will be analyzed to determine whether and how they would fit into the presented scenario of this thesis.

In chapter 4 a distributed hash table as basis for a P2P system will be chosen. This will settle the environment, in which the intended solutions developed in this thesis will work.

The developed abstract model of trust will be presented in chapter 5, which incorporates all previously acquired knowledge. It will start with a description of the developed model and will present the details of the developed implementation.

In the following two chapters the achieved results will be discussed. Finally a summary will provide an overview of the accomplished work and identify it's weaknesses. In the conclusion recommendations for further research will be given.

2 The Notion of Trust

Trust is a very common word. In the course of this chapter different definitions of trust will be analyzed and one of them will be chosen to be used in this work. Furthermore, a common example that stresses the importance of trust in computer science will be presented. After looking at common elements of trust in informatics, the scenario this work is based upon will be introduced.

2.1 Trust in sociology

Trust is something used in everyday live as an element of social interaction. While its use is common as a social phenomenon, developing a definition for it is a complex task as it is purely subjective. When analyzing trust a variety of factors will be found, that in different contexts have different weigths. For example, for a doctor to be trustworthy, his competence has an absolute priority, his loyalty counts less, which may be - for judging a friend - an important factor. Speaking of competence and loyalty, these are already two examples of *sources* of trust, which are based on attributes of other peers. Others include reliability, goodwill, or even the way the other person dresses. Besides this, the subject's own situation and mood make up other factors of trust, so a heavily weighed "human factor" expressing the pure subjective nature of trust, needs to be included.

In literature there are two common definitions of trust, *reliability trust* and *decision trust*. In [Gam88] the first form is defined:

Reliability Trust: Trust is the subjective probability by which an individual A, expects that another individual, B, performs a given action on which his welfare depends.

The former definition stresses the dependence on an action the trusted party should perform and defines trust as probability of this happening. This is a straight forward definition of trust. Imagine a store, were you buy some item. Now further imagine a storekeeper, that shortchanges you every now and then. Being an example, the obvious solution of leaving this store without coming back again, will be ignored. So now we can count the time we have been purchasing at that store and the times we have been shortchanged. Dividing the former through the latter will result in the statistical probability of a negative outcome.

Decision Trust: Trust is the extent to which one party is willing to depend on something or somebody in a given situation with a feeling of relative security, even though negative consequences are possible. [MC96]

This definition includes more elements of human interaction involving trust, by stressing that negative consequences are possible. This *risk* becomes more important, the higher the loss would be. Imagine an online business platform, accepting credit card numbers as the only method of payment. Giving away that number leaves its customers in a vulnerable position, *depending* on the vendor to act according to laws, *risking* an enormous financial damage.

A possible trust decision could therefore be modeled: A factor - depending on the risk - controls whether a transaction is carried out under a given probability of success. For illustration purposes figure 2.1 represents this kind of trust in a pseudo code implementation.

In [Han00] the author names a variety of sources of trust. This is illustrated in graphic 2.1. They are weighed differently by their context and a human "factor". Trust involves both actions from other partys a subject depends on and it's subjective influences. In the course of this work the definition of *Reliability Trust*, will be

Listing 2.1: Pseudo code implementation of Decision Trust

```
enum risk_t = {high, low};
enum trust_t = {high, medium, low};

bool transact(risk_t risk, trust_t trust) {
  if (risk ==  high) {
    if (trust ==  high) {
     return true;
    }
    return false;
  }
  if (risk ==  low) {
    if (trust ==  low) {
      return false;
    }
    return true;
  }
  return false;
}
```

used, because it serves best as a straight forward definition. Furthermore, the concept of risk in a transaction would shift the focus away from a generic concept of trust management, as it would require to check transactions on their possible risk. This may need some explanation. Imagine the eBay system would be enhanced, providing information concerning the risk of an operation. The system would require functionality deciding what risk for the user means, depending on the current product, like a cost-trust ratio. That information is too specialized for a system designed to be generic as possible. It requires some semantic knowledge about the transmitted data. We strongly emphasize however that this work serves as a basis which could be specialized and, by deciding on the type of information the concept of risk could be included.

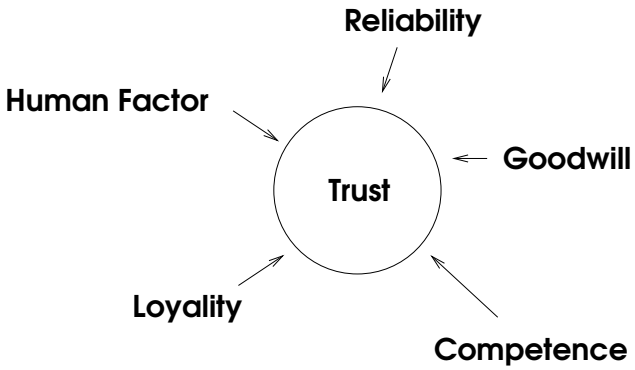


Figure 2.1: Trust accumulates from a variety of sources

The definition of Reliability Trust mentions a subjective probability at which an action is performed. Therefore the probability of an action is defined by the mathematical probability:

$$p_i = \frac{q_i}{n_i}$$

q denotes the successful transactions while n is the number of total transactions with peer i. Of course it is eminent that in the case of $n = 0$ the degree of trust is undefined. In case no decision based on the knowledge of the client is possible, which exactly resembles the scenario's problem of trusting an unknown peer. Additional information is needed. A formal basis for that case can be found in 2.5.

2.2 Social Networks

As a matter of fact, communities develop not only in real world, but in every medium given. These are commonly modeled as social networks. In computer science, this area of research gained high popularity over the last years, not at last because of the rise of standards, which enable expressing relations between entities in a machine readable format. To introduce these kind of networks a definition is needed:

Definition of Social Networks *A social network is a social structure between actors, mostly individuals or organizations. It indicates the ways in which they are connected through various social familiarities ranging from casual acquaintance to close familial bonds [Fou05a].*

A social network can be represented as a graph, where edges represent a "knowing" relationship and nodes reflect individuals. For an example see figure 2.2.

Although these networks have always existed, a fact given by human nature, the concept received attention in computer science lately with the usage of a technique called *Resource Description Framework (RDF)*. This is a standard developed by the w3c [BG99] with the aim of enabling computers to parse information on the Internet automatically. As toady's information on the web is not suitable structured to be processed by a computer this was necessary. These techniques are referred to as the *semantic web.*

One social network, relying on the latter technique is the "Friend of a Friend(FOAF)" Project [1]. While at first sight not trust related, this technique provides an infrastructure for enabling such a system. It provides a method of expressing information about peers and linking that information. Furthermore, it copes with an important topic in the area of trust: Identifying peers. To be able to express information about peers, these have to be identified in a common way, that maps a description clearly to a person. FOAF expresses identity using email addresses. While a person and that person's email address are not the same *you could reasonably assume that all descriptions of a person that included "this person's e-mail address is edd@xml.com" might reasonably refer to the same person.* [Dum02]

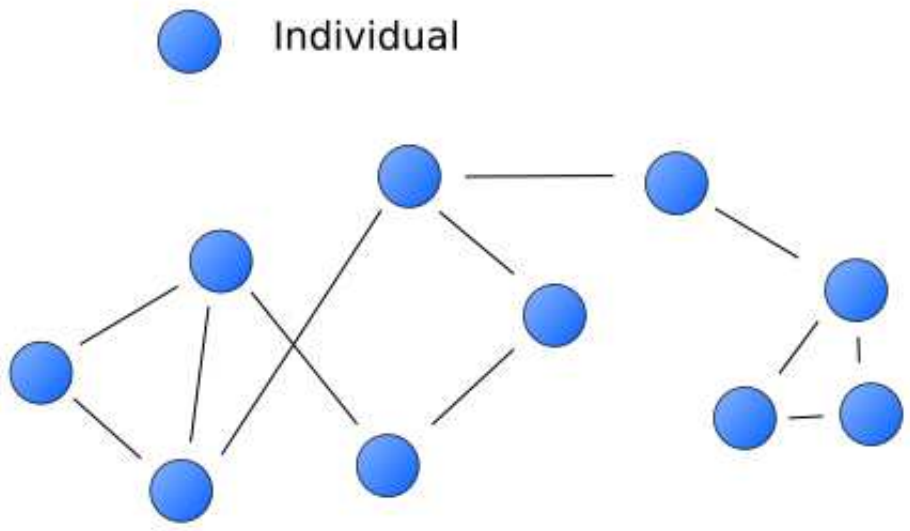


Figure 2.2: An example for a social network ([Fou05a])

Social Networks are used to describe interaction on almost every application. A logical step is applying trust to those, which creates an convenient way, to analyze the trust relations between peers in networks.

[1] http://www.foaf-project.org/

2.3 The Prisoner's dilemma

One of the goals of Computer Science is building an abstract model of reality. Therefore it is not surprising that this import part of human interaction has already received attention. In the area of game theory, the Prisoner's Dilemma was invented as an example that shows *lack* of trust. Therefore, this game is introduced in the following, as trust centric situations can be reduced to the dilemma.

The game has two players A and B. They were captured by the police for having committed a crime, but there is insufficient evidence for a conviction. Therefore, both are separated and each of them gets an offer from the police, for going free when testifying for the prosecution against the other, who will be arrested for 5 years. The downside is, if the other player chooses the same strategy both will be arrested for 4 years. When none takes the offer from the police they go free after 2 years, for the lack of sufficient proof. To sum up the situation:

	Prisoner A silent	**Prisoner A betrays**
Prisoner B silent	Both 2 years	A: Free B: 5 years
Prisoner B betrays	B: Free A: 5 years	Both 4 years

Table 2.1: Rules of the prisoner's dilemma

The optimal solution for both players is to cooperate. If they stay silent they will receive a moderate punishment of 2 years. But, from a game theory's point of view, the optimal strategy for every player is to betray. If the other player stays silent, he will be free, otherwise he will have one year less of punishment. The dilemma is that both players can not develop a strategy, as they are separated and even if there were a cooperation no prisoner knows if the other will not betray. Here we have a clear lack of *trust* .

The game has a Nash equilibrium, which describes the core of the dilemma. The only strategy, in which no player has a gain in changing his own strategy is the "both cheat" situation. The obvious "both cooperate" case, leaves every player the opportunity to cheat and receive a gain.

If the game is played repeatedly, there are different strategies to play this game, but first an abstracted form of the game needs to be introduced.

		Player B: **b1**	**b2**
Player A:	**a1**	+5 / +5	-5 / +8
	a2	+8 / -5	-3 / -3

Table 2.2: Abstracted rules of the prisoner's dilemma

In this game both players have an abstract "choice", receiving points for their actions in each iteration of the game. Now we see more clearly that we deal with a non null-sum game, as it is not the fact that in every outcome a player gains as much as the other looses. There has been a large number of research carried out on the dilemma and different strategies arose: To name only two there is the forgiving strategy, that always plays nice, hoping for the other player to do the same. The "nice" strategy, starts playing nice until the other one betrays, sanctioning the other player and returning to a "nice" strategy after the other cooperates. It is also a forgiving not vengeful stategy. The latter also pays out most, as it prevents revenge and relies on the opponent to play nice in the end.

Hofstadter's Modifications There have been multiple modifications of the game. We will have a look at the version of Douglas Hofstadter [Hof85], which was designed to reflect business habits. Now both players have a bag and exchange them. Both agreed previously that one bag contains a product and the other contains the payment. But every side can also handle an empty bag. Again all strategies applied to the prisoners game can be applied. The chosen act will boil down for both players to the same outcome. One is receiving a **short high gain** by receiving payment for no product, or not paying for a product. This will of course stop further interactions. The other is a **longstanding moderate gain** by behaving correctly. This leaves the possibility to do further business with that customer.

The Prisoner's dilemma and especially the modification of Hofstadter serve in the case of the Internet as business platform as a very good model. It has already been proven in reality and is now stressed by the special attributes of the Internet. Most people even today believe that there is a certain kind of anonymity on the Internet. This misconception produces a relative feeling of security that in turn increases the probability of betraying.

Most striking however is the fact that on the Internet, especially in P2P systems, there is a high degree of transaction between people who have never met an will likely not meet in the future. When facing a possibly anonymous vendor, that is unknown to a peer, giving away credit card information is a sensitive task. It can therefore only be hoped, that the other player will play using the "nice" strategy, as a fair player will be the one having loss.

Pavel Watzlavick notes in his book "How Real is Real?", that *even* if the prisoners can communicate and settle an agreement, they will need to *depend* on the other being cooperative and the dilemma starts again:

> "*each [prisoner] will invariably realize that the trustworthiness of the other depends largely on how trustworthy he appears TO the other, which in turn is determined by the degree of trust each of them has FOR the other – and so forth ad infinitum.*" [Wat76]

2.4 Trust in computer science

The Prisoner's dilemma already introduced the importance of the topic. While previously different definitions of trust in sociology were given, computer science offers a broad spectrum of trust applications, involving different subjects and appliances of their trust relation. *Blaze et al.* describes the duty of a trust management system in their work as follows:

> *"Does the set C of credentials prove that the request r complies with the local security policy P?"* [BFL96]

Blaze uses the system for authorisation. Therefore some more elements need to be added, to have a sufficient set of trust elements:

Entities These are the basic acting elements. There can be different kinds in trust interaction. They include humans interacting via a computer, but may at the other extreme be purely digital entities, as for example a X.500 directory, that interacts with a X.509 certificate, as chapter 3.1.3 will show. Another common name is *principals*.

Credentials Associated with Entities are sets of credentials which describe them and their relationships. A classic *Access Control List (ACL)*, as the UNIX^{TM} style system, describes possession of files by individuals. Chapter 3.1.5 will introduce certificates describing membership in groups and the ability to delegate trust as a set of credentials.

Authenticity Both previously mentioned systems form the basis of each and every trust system. Entities and their Credentials are the basis for all decisions. Therefore it is critical, that the peer can *prove* its authenticity. As cryptography provides this functionality, its importance in the area of trust systems needs to be stressed. It will receive attention in chapter 3.1.1.

Policies The authority which decides about the validity of the request and comes up with a trust decision. Policies can be distinguished, depending on the type they operate on. In [KR97] the authors define three types:

- Principal-Centric Policies: Describe who is allowed to access resources at a given privacy level
- Object Centric Policies: Commonly a ACL style policy. Each object is associated with a list of authorized users.
- Action Centric Policies: Bases decisions on which action is to be performed by a given "token".

Relation On behalf of a policy relations are established. Fundamentally any kind of interaction between peers can be a relationship. Once again it includes a wide range of possible appliance. After a successful authorisation a peer could be certified by another, or depend on another. The web of trust serves here as an example. After a meeting in person, both parties certify each others keys, to be a valid identifier of the person. The web of trust will be explained in chapter 3.1.4.

Trust Classes The resulting trust, can be divided into classes, given the involved credentials and policies, as described in [AJ05]:

- Provision Trust: The party's trust in a service or resource provider. This kind of trust is commonly encountered when being prompted to install digitally signed ActiveX applets, when updating a windows installation.
- Access Trust: Describes the trust a party grants others when allowing access to resources. ACLs are a closely tied concept to this kind of trust.
- Delegation Trust: The trust into a party, to act and decide on the behalf of the other party
- Identity Trust: The trust that an agent is who he claims to be.

To sum up a trust system, as seen in figure 2.3 may be seen as a system, that lets other nodes join after fulfilling a set of constraints (e.g. a successful authentication). After joining the system, a peers credentials will be evaluated by a set of policy rules. These define the relations systems have with each other, that in turn authorize the use of resources and allowed actions.

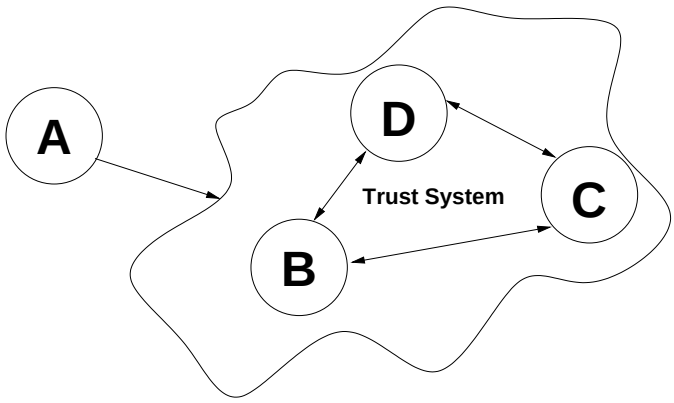


Figure 2.3: A trust system with three nodes B,C and D. A needs to join the system, to build relations

2.5 Developing Trust by Reputation

In the previous chapters different sources of establishing trust were introduced. A closer look reveals that these have been subjective measurements. However, a person normally does not totally rely on his own impression of a person. A very common source of trust is what other - possibly already trusted - people think about others, especially when the third party is unknown. The previous experience of others will be used by a person, to

fill in a gap in his knowledge. For example when searching for an online store for the first time, it is common to ask others for their recommendations. This information will be weighted by giving a high importance to recommendations of already trusted persons, or simply counting the number of positive votes for a certain shop. Following this step, a trust decision will be produced, influenced by the previously collected data. A possible source of trust to somebody or something is therefore a good reputation. Before going into details a definition will formally define this process.

Reputation: Reputation is what is generally said or believed about a person's or thing's character or standings. [AJ05]

This definition stresses that reputation is a kind of collective measurement of trust. While trusting someone usually means that this person has a good reputation, the reverse is not true, expressing only a "general" view.

It is important to stress the difference between trust and reputation: [AJ05]

- *I trust you because of your good reputation*
- *I trust you despite of your bad reputation*

Using reputation information as basis for trust decisions seems to be sensible. It was already stated that problems arise when encountering a unknown peer, because they cannot be judged using normal social style interaction. Therefore the rather common task of collecting information from "other sources" will be used, thus discovering the reputation of that peer for making a trust decision. Using this method, an interaction is based on the experience other users had before with this party.

A reputation system is therefore a way to enhance a trust system. Given a set of peers that are already in a trust relation and another peer that has a trust relation with one of the other peers, a reputation system helps establishing a trust relation between previously unknown peers, as illustrated in figure 2.4.

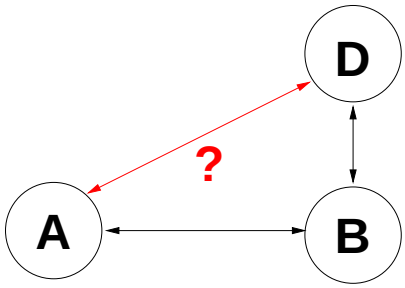


Figure 2.4: A reputation system used to establish a trust relation between node A and D by using the trust relations between (A,B) and (D,B)

Reputation in sociology enforces a system of social control. Because it is a publicly available measurement of trust, misbehaving clients can no longer hope that their acts will be "not noticed". The availability of information will therefore put pressure on them as their acts are available to everyone. The information can easily be extended to a "cooperative sanctioning systems" based on reputation.

For a common example of reputation, examine a popular fairy tale: A sheep guard was bored, as nothing happened all day. That is why he yelled "wolf", pretending being in danger. When everyone came to help him, of course there was no such danger. He repeated his game so often, that when there was a real wolf no one came to help him.

This fairy tale illustrates how deep the concept of reputations is integrated into everyones social lives, leaving an advantage to "honest" peers with a good reputation and punishing the dishonest. The sheepguard managed to have such a bad reputation that no one trusted him any longer.

Reputation Systems and Trust Systems *Jøsang et al.* sum up the main differences between a reputation system and a trust system, as the difference between a subjective and a collective measurement. Furthermore, the measurement of trust varies. While trust systems take into account subjective and general measurements as factors for trust decisions, reputation systems rate information about specific events, like transactions. Also when making a trust decision, a reputation system needs to assume some kind of transitivity of trust to come to a trust decision, while in an trust system this is an explicit action.

Reputation Systems Commonly these kind of systems are divided into two categories. A *centralized* scheme aggregates the reputations at a central location and returns the scores to participants, when needed. Examples include the eBay systems or Googles PageRank. These systems will receive attention in chapter 3.2. The second type are distributed systems. Each client maintains a own repository of experience and returns it, when needed. In that case a client seeking for that information has to actively aggregate it from the clients. The P2PRep solution from *Samarati et al.* [DdVPS03] and the Eigentrust algorithm of *Kamvar et al.* [KSGM] will be presented in chapter 3.3.

System Design In this work it was decided, to design a distributed reputation system. These systems not only fit into the application scenario described in the introduction and will elaborate in chapter 2.7, but are both more reliable and efficient, as by distributing the information across a network a “single point of failure” is eliminated. These system are further not “owned” by s.o. that could possibly influence the information in some way.

2.6 Trust Metrics

In chapter 2.5 it became evident that a reputation system is a suitable way to establish trust relationships. Obviously reputation is a complex topic that yields more questions about usage. This will be illustrated in the next paragraph.

To formalize the concept of reputation it can be thought of in terms of mathematics. Let A,B and C be peers and let $\circ$ be a binary operation expressing trust. The question is whether trust has a concept of *transitivity*.

$$(A \circ B) \wedge (B \circ C) \Rightarrow A \circ C \tag{2.1}$$

This means that if A trusts B and B trusts C, is it true that C trusts A? As this work’s concepts are based on reality, the decision is not obvious to answer yes, but it is *likely*. The area of research touched with this question is huge and centers around *social networks.*

Definition Trust Metric *There are three inputs to this trust metric: a directed graph, a designated ”seed” node indicating the root of trust and a ”target” node. We wish to determine whether the target node is trustworthy.*

Each edge from s to t in the graph indicates that s believes that t is trustworthy. The simplest possible trust metric evaluates whether t is reachable from s. If not, there is no reason to believe that t is trustworthy, given the data available.[Lev04]

Note that Trust Metrics are also referred as “Reputation Computation Engines” in literature.

Given a set of four nodes A, B, C and D and a set of trust relations as seen in Figure 2.5, the Question trust metrics center around is whether to trust D and to what extent. There are different metrics, that refer to that problem. A linear approach called “linear pool” was presented in [Gen86]. Another one is the noisy OR [Pea88] by Pearl. Further elaborate examples will receive attention in chapter 3. Trust Metrics and Reputation Systems have not only application domains in apparent situations, as encountering peers, they also are the underlying technology for Googles PageRank system [BP98].

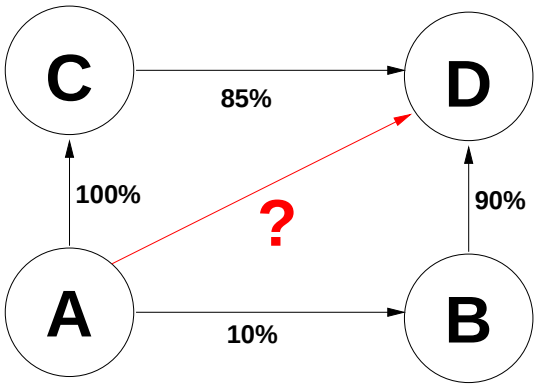


Figure 2.5: A sample trust metric with a trust root (A) two trusted parties (B,C) and an unknown party (C)

Given the research on trust networks, there has also been research on possible attacks on these in [Lev04], [Dou02]. After defining the general surroundings of trust, the next chapter will present the specific application domain of this work, upon which the scenario is based.

2.7 Scenario on Distributed Trust Management

Trust and Reputation systems are in widespread use today. The first use of Access Trust is the trust system by Balze, that will receive attention in chapter 3, was introduced in 1996. The Public Key techniques, which form the basis for every kind of Identity Trust, date back to 1976. With the rise of the Internet, online platforms caught up and used trust models as oriental guide for customers. While the solutions seem sufficient for most applications, another area, where the lack of trust has not been addressed sufficiently, can be identified: The area of P2P systems. These distributed networks require virtually no authentication and even actively help users to hide their identity. For those two reasons these systems have been abused for sharing copyright protected material. This work, of course, wants in no way to support any kind of copyright violations and distances itself from it.

However when a user chooses to download a piece of information from a peer to peer network, let it be a CD image of a popular LINUX distribution, how can he be sure to download a correct copy of the distribution and not a tampered image? Cryptographic hashes usually assist the user in his decision, but what if there is no such extra information? The peers, that share the image will unlikely be known to the user, therefore there is little assistance from that source of information. The peer might end up downloading an unknown file.

A reputation system might help in that situation. If the user lacks trust in the services offered by other clients, he lacks - as referred in chapter 2.4 - *Provision Trust.* However there is no direct trust relation between the user offering and the peer searching for information. Therefore a user needs to come to a decision based on a collective measurement of trust, which is in turn a reputation system. The user could query for the reputation of the offering peer and base his decision on that.

Designing a reputation system, that fits the needs of a P2P system, especially the needed flexibility is a challenging, but feasible task. P2P features a large userbase, that often encounters unknown peers. The openness of these systems makes them different to other existing trust systems. While normally every system has to fulfill some constraints to participate in a trust system - let it be an authentication after a registration process - P2P systems are open and usable for every user. Therefore a reputation system needs to handle every client and also resist every client's attacks. P2P research has been designing algorithms to be faster in locating information in a vast distributed network, which is a problem that every distributed reputation system also faces.

This work will present a reputation system that is designed to fit the needs of P2P systems, by integrating it into a core protocol. Recommendations will be retrieved by employing a distributed hash table, chosen in chapter 4. The table will retrieve a key of a node as input and provide the key of a node managing the inputs nodes as output. The returned node will provide a number of votes collected from all peers, that want to express their

recommendation. The integrity of votes will be guarded by a digital signature, attached to the vote. For this purpose, the system will use a decentralized private key infrastructure, that will provide the required public keys. The system will work under the assumption that the majority of voters will provide a reliable basis, while weighting their input on the basis of their previous performance. To get a closer idea of this sketch of design, the next chapter will introduce existing solutions for enabling trust. Based on components of these solutions and on a platform, evaluated in chapter 4, the system will be presented in chapter 5.

3 State of the Art

In the previous chapter the scenario of this work has been described. This chapter will introduce the current state of research. Different solutions, that can provide valuable input to the design of this thesis have been implemented during the years. As it is intended to collect information from different peers about the reputation of a third peer, mechanisms for identifying these need to be found. Furthermore, it was postulated, that this system needs to be resistant to attacks.

Commonly in computer science, when the validity of s.o. or s.t. needs to be ensured, cryptographic methods are chosen. The first subchapter will research existing solutions in that area.

Furthermore, as a reputation system needs to be designed, both chapter 3.2 and chapter 3.3 introduce existing systems. The first chapter focuses on the central architectures. While these systems do not meet the design goal of building a distributed system, a number of very capable trust metrics have been developed.

In subchapter 3.3, two distributed reputation systems will be closely analyzed. The system which fits best the needs of the scenario will be chosen as basis for this work.

In the last chapter, the security research on reputation systems will receive attention, which enables an analysis of the the potential risks and attacks to the system.

3.1 Cryptography based solutions

Every possible trust solution today relies partially on features provided by cryptography. The solutions presented in this chapter address the problem of *Identity Trust.* As defined in chapter 2.4 the presented solutions will enable another entity to trust in the authenticity of the peer. It could identify itself using the presented techniques. As an introduction this chapter starts with a brief overview on encryption, to refresh the knowledge of the basic functionality and then have a look at further developments.

3.1.1 Asymmetric Encryption

Historically encryption was a difficult task because of the need to exchange the key before being able to use encryption, as the symmetric key - used for encryption *and* decryption - had to be known to both parties. Yet it would have been impossible to create a lager scale system of many different nodes, as the key management would have overtaxed the system. A trust management solution would not have been possible. Public key cryptography, also called *Asymmetric Encryption*, described in a paper by Whitfield Diffie and Martin Hellman [DH76], solved the problem of key exchange. In contrast to the symmetric approach where there is only one key, the encryption and decryption keys are separated into a private and a public key. The private key, also called *decryption key,* is used to decipher messages encrypted with the corresponding public key.

This technique creates and enforces the important basic features of transaction: *Privacy and Authenticity* . Privacy guarantees that only the receiver of the message can decode and read the message. The latter enables a recipient of a message to be certain about the identity of this communication partner, as normally only the key holder can use the key, with which the message is guarded. When using a digital signature the feature of *Integrity* is also provided. Integrity prevents data change during transmission.

For this work it means, that both the integrity of the voting information and the voter's authenticity could be proven. This is a fundamental need for a system that establishes trust on information received from third parties, as otherwise this information could have been altered during transmission.

Some examples for Public Key techniques are:

- Diffie-Hellman
- RSA encryption algorithm
- ElGamal

The RSA algorithm, described 1978 in [RSA77], works according to [Inc05] as follows:

Upon key generation two random prime numbers and their Product $n = p * q$ are generated. Both primes have to be big although significantly differ in dimension. Furthermore, let there be a public exponent e and a private one d, that comply with the following rules:

- $e < n \ \wedge \ ggt(e, (p-1)(q-1)) == 0$
- $(ed-1) \ mod \ ((p-1)(q-1)) == 0$

The primes p and q have to be deleted after finding e and d.

- A message m can be encrypted by using $c = m^e \ mod \ n$
- A message c can be decrypted by using $m = c^d \ mod \ n$

An important part of this work is the *Integrity* feature. But first the motivation for cheating a trust system will receive attention. The scenario is based on the fact that peers lack a lot of meta information on the Internet when deciding if it is safe to transact with a given peer. This work therefore wants to provide a trust system to support that decision. Given the evidence that a possible hostile peer is encountered, the possibility to cheat that system needs to be eliminated, because otherwise this would lead or the system ad absurdum. A possible attack to such a system would be to tamper the trust information. A client requesting reputation information for a given peer could easy be mislead in its trust decision if the published information could be altered. This is why the integrity of the trust information needs to be ensured. This is done by providing a digital signature of the data. This requires further information:

Digital Signatures enable their users to assure that the signed message has not been tampered in any way during transmission. It is achieved by creating a unique hash of the message. To prevent the hash itself from tampering it is *encrypted* with the senders private key. The resulting checksum will be attached to the message. The recipient can validate the message by computing the hash of the message once more. The hash attached to the message can be *decrypted* using the public key of the sender. If both hashes are equal, the integrity of the message has been proven, as seen in Figure 3.1 on page 15.

Furthermore, the transaction could take place on encrypted channels. Although the information is public available and already equipped with a signature, the subject of the reputation information should be hidden from a possible attacker. By hiding the content of the transaction from others, attacks on the trust information of specific peers could be prevented. Also the Authenticity information is of great value. Every trust system is useless if an attacker could pretend to be a trustworthy peer and circumvent the security mechanisms.

Disadvantages of Public Key Cryptography include higher computational costs, besides a pressing disadvantage, that is not a problem of the concept, but limits in practice the level of security: previously it was mentioned that Privacy, Integrity and Authenticity are supported. In practice however the key management remain a problem. If an attacker manages to get hold of the private key, he could impersonate the key holder. Common attacks to rob a private key are simply breaking into a computer, or a “man in the middle” attack during the key exchange. Using the latter technique the attacker does not even need to hold any peers private key, as he simply replaces the peers public key with his own key.

After getting familiar with the basics of public key encryption it is now possible to move forward and inspect trust system, that are built basing or using the services provided by this fundamental technique.

3.1.2 Trust Centers and PKI

The advent of Asymmetric Encryption reduced the importance of key exchanges, while other problems of both asymmetric and symmetric encryption remained. This chapter will present solutions found for managing the life cycle of the key pair. PKI stands for *Private Key Infrastructure*, which as the name already hints, provides means of managing public keys. In a centralistic approach the heart of the PKI is a trusted instance, the *Trust*

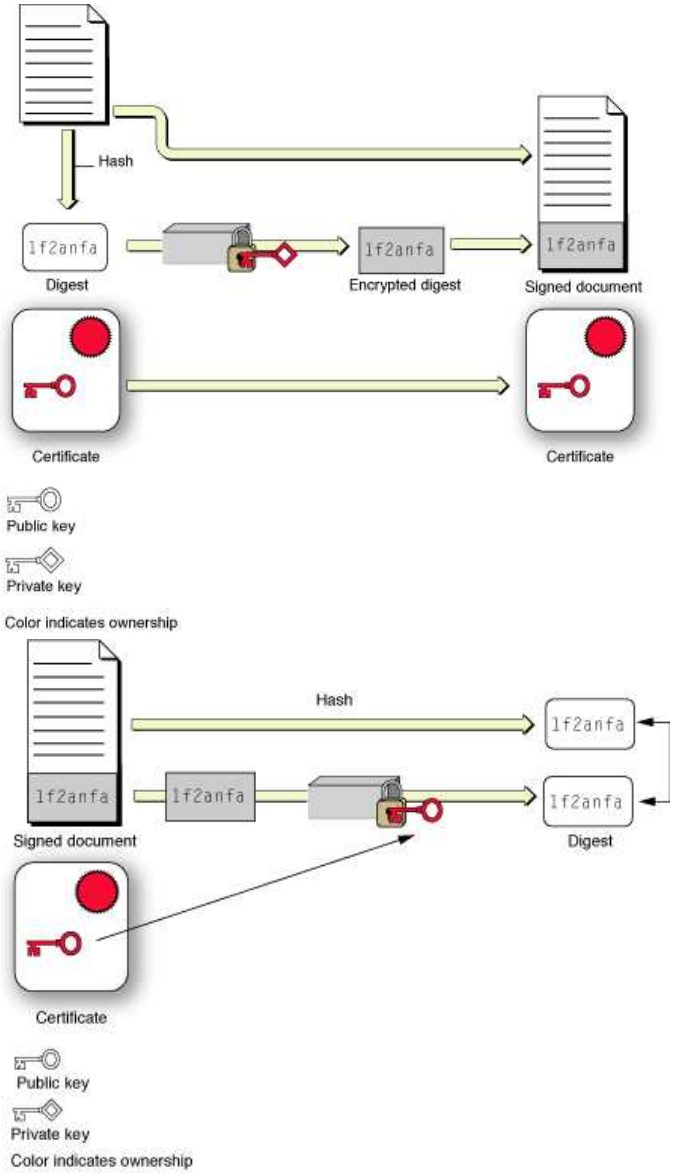


Figure 3.1: Creating and verifying digital signatures [AC05]

Center that serves as central authority managing certificates and keys. It is responsible for key distribution, providing a central place to retrieve public keys, or if a key can no longer be trusted, the key can be revoked. When a key expired, actions can be taken. Given this set of features the trust center serves as instance to validate keys. For example a web browser may request verification of a certificate issued by this authority. Furthermore, the validity of public keys can automatically be checked.

The decentral pondain is a self managing network of public key distribution, that relies on a set of servers for key retrieval.

This management of life cycles of certificates and keys from creation until expiry enables to more secure authenticity, as now a user can be mapped to a given key and also undo that mapping with a reasonable amount of work.

In the following two examples of a public key infrastructure will receive attention.

3.1.3 X.500 Directories

The X.500 directory standard was developed as a joint project of the CCIT and the ISO committee. Both institutions started researching in 1984 separate solutions that were in term merged. The first standard dates from 1988 [CCI88b] . The directory was designed to hold objects and their attributes of different kinds, including information concerning systems, organisations and people. Per directory, the concept follows a centralistic approach, storing information in one place. The directory is represented as a tree, whose root represents the root object, like an organisation. Each subtree below the root contains finer grained units, following the organisation example e.g. a department. Leafs could describe employees of the department. An attribute of such a leaf could be the email address of that person.

Particular interesting from the cryptographic point of view are the X.509 certificates [CCI88a], which were developed with X.500. There were primary designed as a an access control mechanism, so that users could authenticate to alter their data in the X.500 directory. A certificate binds a user with a particular *Distinguished Name* , email address or other credentials to a public key. The Standard also includes ways of revoking certificates via *certificate revocation lists.*

Fields of a X.509 certificate include:

- Issuer: The Certificate Authority that signed this Certificate
- Subject
- Subject Public Key Info:

  Subject Public Key
- Signature Algorithm

While the X.500 standard was never fully implemented X.509 remains a standard for certificates. For this work it is of course interesting to have a look at the trust metric propagated of this model:

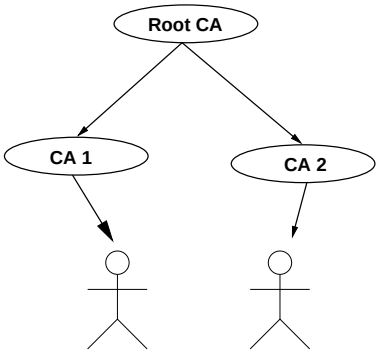


Figure 3.2: Trust relation between X.509 certificates. If the root CA is trusted, all certificates are trusted too

The Trust Center serves as *Trust Root.* Every node that is known to the root is fully trusted in the system. Therefore, figure 2.5 from chapter 2.6 should be altered in the following way: Let A,D be a local client. C is the Trust Center. The resulting trust model is described in figure 3.3. Obviously the trust root is no longer inside the client (A) that tries to judge trustworthiness, but is now transferred to a central instance (C). As that instance is considered totally secure, client D is considered trustworthy by providing conformity to the PKI policies. In this example trust is *transitive.*

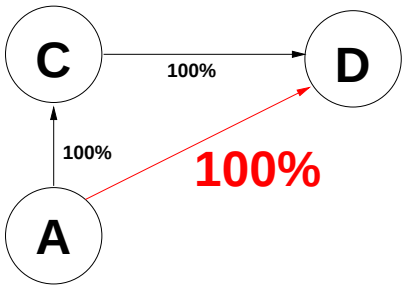


Figure 3.3: Trust metric with a trust center (C). Ultimate trust applies to all members of the PKI

A closer look reveals some imminent shortcomings for a trust system.

- Central Trust Management Approach
- Full trust in the Central Authority
- No possibility to express distrust

The system suffers from problems every central system has. A central concept provides a single point of failure, which could in case of an attack be exploitable to denial of service (DoS) attacks. Furthermore, if a malicious peer manages to have its key signed from the authority, other peers assume it is trustworthy.

However, this technique describes a mapping between a person and a cryptographic key, which is signed by a trusted third party. Under ideal circumstancesi, the goal of authenticity has been reached. This can be used to base access control features on top of that concept.

3.1.4 Web of trust

In the previous chapter, a way of binding a subject to a certain key by using a trusted root certificate authority, was introduced. While this may fit for most needs, a both trivial and powerfully approach is the transfer of social live into virtual space. As humans always tend to find contact to others there has been a rise of virtual communities, where anonymity is overcome by getting to know each other on a virtual basis. These communities form the basis for the “Web of Trust”.

The principle is rather simple. Users receive public keys of other users commonly in electronic form from a keyserver. These servers store the public keyring of a user. To ensure the authenticity of the received public key, it is necessary to compare the key, or more specific it’s fingerprint, with the original key. This can be done by meeting the owner in public, e.g. at special “key signing partys”, or by comparing with a printed version, received from the owner of the key on some other channel, e.g. printed on his business card. After checking validity, the key is signed, expressing the successful completion of that process. As it is not possible to know every communication partner in person, it becomes necessary to rely on others, that have already checked the key and testified its authenticity, which is a form of *trust.* By uploading the signed key back to the keyserver, a user provides his validity information to others.

The binding is ensured to be correct by other users that prove the correctness. However, there may be malicious users, that aim at circumventing the system, by incorrectly verifying other user’s keys. This could easily lead to a trust relation about the binding of a key to an identity, which is incorrect. Therefore a trust system is employed. Users can rate other user’s ability of correctly verifying other user’s keys in five different levels:

1 = I don not know
2 = I do NOT trust
3 = I trust marginally
4 = I trust fully
5 = I trust ultimately

Note, that the word trust is hereby used to mean trust in an owner and trust in a key. The trust in an owner information is kept private, the trust in a key information is expressed by signing and uploading that information to a keyserver. The policy, that decides, whether a given key is valid yields a positive result if the following conditions are met:

1. it is signed by enough valid keys, meaning
   - you have signed it personally
   - it has been signed by one fully trusted key
   - it has been signed by three marginally trusted keys
2. the path of signed keys leading from the key back to your own key is five steps or shorter

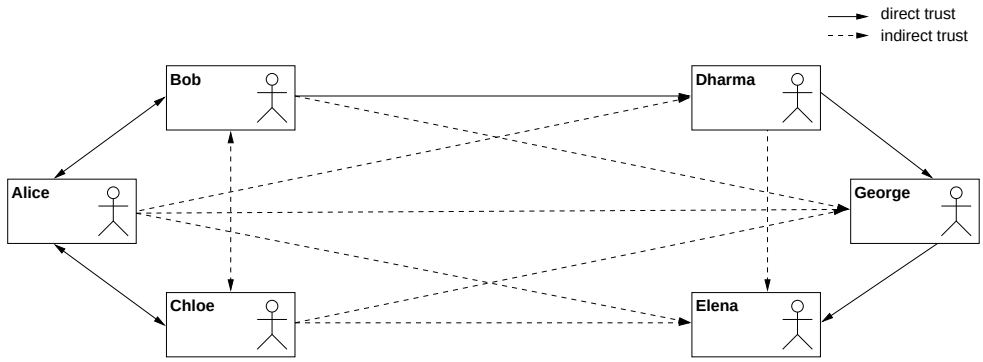


Figure 3.4: Trust relations in a Web of Trust. Starting from Alice a chain of direct trust relations enables an indirect trust relation between Alice and George

This approach moves away from the central structure of few Certificate Authorities to a more public model. Users are not forced to have ultimate trust in a certificate authority they only know by name, but can rely on their social contacts to support their trust decision. In fact every user is his own trust center, issuing his own certificate and relying on a trust chain he defines. Everyone can start a web of trust, as these can coexist and can be linked by member in both “groups”.

3.1.5 Decentralized Trust Management

Another approach directed at establishing *Access Trust* was done by *Blaze et al.* in [BFL96]. The goal was to replace existing ACL solutions, whose main problems were described according to [BFK99]:

- Authentication: While it is convenient for local only computers to have knowledge of every user accessing the system, it is not the case in a distributed system. Therefore, a new model had to be developed allowing to unify Authentication and access control.
- Delegation: As large distributed systems tend to suffer increasing complexity, administrative tasks need to be decentralized. This allows more than one administrator. In the past these mechanism were implemented using Access Control Lists (ACL), the Unix systems typically used group permissions to handle this need, which add unnecessary complexity to the system.
- Extensibility: The traditional ACL systems were not designed to be extensible. The common distinction between the owner of a file, members of the files group and all others is not scalable.

- Local trust policy: A concept not known to traditional systems are all kinds of trust polices and the relations between other systems. While it is now possible to build some kind of trust between machines by adding their public keys to the local trust policy more complex interactions - for example the mentioned *transitivity of trust* - are not possible.

A typical workflow for processing a signed message was introduced in its paper as:

1. Obtain certificates verify signatures, possibly. determine public key of signer
2. Verify that certificates are unrevoked
3. Attempt to find "trust path" from trusted certifier to certificate of public key in question
4. Extract names from certificates
5. Look up names in databases that maps names to the actions that they are trusted to perform
6. Determine whether the requested action is legal, based on the names extracted by certificates and whether the certification authorities are permitted to authorize such actions according to local policy

The papers by Blaze suggests, to replace steps 3 to 6 by submitting all data together with a local trust policy to a central service. The service is required to form a trust decision based on the given data. The question *"is user A allowed to perform the task B?"* is replaced by *"Does the set C of credentials prove that the request r complies with the local security policy P?"* which is handled by a *Trust Management Engine"* providing a result in the context of (r, C, P).

A prototype implementation of a trust management engine called PolicyMaker was presented in the first paper of Blaze. In [BFK99] a second engine called *KeyNote* was presented. Both implementations differ in the architectural boundaries that were drawn.

First to note is that no longer only humans are accepted as entities. The mapping of a cryptographic key to a real name was replaced, as the system checks only the validity of the given *key,*which is the principal of this trust solution. The system is enabled to be layered on top of already existing PKIs, including X.509, as it relies on cryptography for providing the trust root. Another aspect is the type of the policy, which is *Action Centric* as defined in chapter 2.4: *The PolicyMaker system provides a simple language in which to express conditions under which an individual or an authority is trusted, as well as conditions under which trust may be deferred.* [BFL96] The general nature of this concept does not explicitly implement a trust metric, but does implicitly allow to build one based on the programability of polices.

By enabling an application to specify a policy, the burden of enforcing and creating policies is passed away from a central instance. This enables flexibility to implement any kind of trust metric that is suitable for the needs of a system. Furthermore, the locality that this approach represents avoids *the need for the assumption of a globally known, monolithic hierarchy of "certifying authorities"* [BFL96].

For a better understanding, imagine a web browser that commonly accepts SSL keys, that are under a month outdated, without normal user interaction. But when receiving an outdated key from a online banking system it would warn the user and refuse connection to that system. This example shows the support of a special *context* of operation, that could be supported by a programmable policy. As different applications and even the programs may require different trust decisions, depending on the situation, this concept is highly promising in the trust research.

3.1.6 Conclusion

The solutions presented in this chapter, as already stated in the introduction, are systems providing Identity Trust, with the solution by blaze shifting focus towards Access Trust. They will form an important basis, as every trust solutions needs to layer itself on top of an identity trust solution, as already stated in chapter 2.4. They are, however, not sufficient for our work, as it aims ultimately at a solution enabling Provision Trust in a P2P system.

3.2 Central Reputation Systems

A variety of reputation solutions, relying on a central instance exist already today in mostly web based algorithms. The classic example of such a service is the eBay system, that will be described in the following. Other solutions use elaborate algorithms to provide the system with a better trust recommendation. The systems described in this chapter are Google and Advogato.

3.2.1 eBay

One of the most common reputation systems on the web is built in the form of a "feedback forum". This technique is widely used and presents users with the possibility to express a vote at the end of a transaction. But first recall the "prisoners dilemma": Without any kind of reputation system there would be both for customers and vendors virtually no reason to buy or sell products. Vendors would have little reason to sell high quality products, as even if there is sanctioning, few other potential customers will be informed about it. So it would be highly probable for customers to buy a pig in a poke. Vendors that provide quality products would not be able to match the prices of the betrayers and customers would not be able to distinguish them from the others. This would destroy the market.

eBay The Internet business platform eBay www.eBay.com will serve in the following as an example. The platform provides an infrastructure for managing, buying and listing auctions articles. Being - simplified - just a platform for listing items, there is of course no warranty of any kind for products sold via its interface. Much research has been carried out on eBay because of its reputation system, that is considered successful in a market as *"ripe with the possibility of large-scale fraud and deceit"* [Kol99].

In [RKZF00] the author exploited 3 key features for reputation systems:

- Long living entities: provided for future interactions
- Feedback captured and distributed: stored for reference
- Feedback guide buyers decision: use it

While the features defining a reputation system were extracted problems were also observed in the eBay system. As most systems depends on the "quality" of their service reputation systems stand and fall with the quality of the reports. It seems trivial to note that reports have to be done, the asynchronous "workflow" of an auction platform hinders this. To file a report, the outcome of the transaction will be judged, but the system has to wait until the transaction really finishes. On an auction platform, this includes big delays resulting from the individual transaction partners and the parcel service. So expressing a report can be forgotten or simply is considered too much work. It must of course be noted that in the case of a bad outcome the voting mechanism will more likely be used, but by judging only bad outcome the quality of reports is lowered significantly. Another form of lowering the quality leads to the second point: Reports must be honest. When reports are truly wrong few techniques can be used to filter those. With most existing clique detection algorithms needing a huge amount of input, these will be *in dubio* inaccurate. As eBay's system allows feedback on feedback, there is also the possibility of getting punished for giving true reports. Therefore most reports tend to be neutral or good for the sake of avoiding negative consequences. Of course, this way the data that can be obtained is mostly irrelevant, as it does not reflect an objective expression.

One last problem, that must be noted is the limited scope of common reputation systems. Appliance is mostly limited to the target platform it was designed for. If however there is a technical possibility for interoperability the "political" problem have to be considered, as the owners of these systems have little interest in cooperation. More common is that it is tried to make the features of their systems unique by trying to patent them. Upon receiving a patent, methods of enforcing these will more likely be used than cooperating.

From a trust metric point of view the eBay system leaves the decision, of whom to trust totally to the user. The system provides all information about all peers, so the user is theoretical enabled to check the trustworthiness of every voter. The system aims at enabling Provision Trust, which in turn relies on Identity Trust. The latter is

ensured by using a password based identification scheme, which leaves the door wide open to attacks ranging from brute force to phishing.

3.2.2 The PageRank Algorithm

A common principle of search engines is based on a trust decision. The rank of a page in the list of matches is computed using a reputation like concept, that bases its decision on the amount of pages pointing at it. The PageRank is therefore an expression of a trust relation, of the entity search engine, in the entity website. As Google puts it:

> *In essence, Google interprets a link from page A to page B as a vote, by page A, for page B. But, Google looks at more than the sheer volume of votes, or links a page receives; it also analyzes the page that casts the vote. Votes cast by pages that are themselves "important" weigh more heavily and help to make other pages "important."* [Goo04]

The described recursive algorithm uses votes as reputation information and weights it according to the reputation of the voting page and the number of votes. The original paper on Google, found in [BP98], describes the behaviour as emulating a "random surfer". The probability of finding a page intuitively increases, when a page is referred by many other pages and if only few links have to be followed, beginning from a starting page. To sum it up, two credentials of a website exist: Distance from a given starting page and number of referrals from other sites. Google started out as an academic research project and is now one of the major search engines available on the Internet. As the project is now on a commercial basis, further research on the PageRank algorithm is kept private. The original paper defines the PageRank (PR) of a page A, that is referred from pages T_1 to T_n, as follows:

$$PR(A) = (1-d) + d(PR(T_1)/C(T_1) + \cdots + PR(T_n)/C(T_n)) \tag{3.1}$$

d is a constant damping factor, that is set to 0.85, expressing the probability that a surfer will further browse this path. C(A) is defined as the number of links going out from page A. Further development on the Algorithm is not published, therefore a complete trust metric for Google cannot be presented, but known facts can be summed up in figure 3.5.

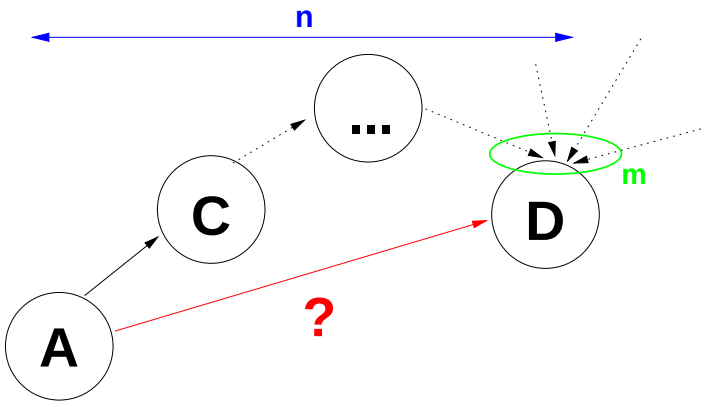


Figure 3.5: The Google trust metric combines two credentials: The length of the path (n) from a trusted starting point (A) and the number of votes (m)

The algorithm also has a description in linear algebra. PageRank values are the entries of the dominant eigenvector of the modified adjacency matrix. [Fou05b].

$$R = \begin{pmatrix} PageRank(p_1) \\ PageRank(p_2) \\ \vdots \\ PageRank(p_N) \end{pmatrix}$$

$$R = \begin{pmatrix} q/N \\ q/N \\ \vdots \\ q/N \end{pmatrix} + (1+q) \begin{pmatrix} l(p_1,p_1) & l(p_1,p_2) & \cdots & l(p_1,p_N) \\ l(p_2,p_1) & \ddots & \cdots & \\ \vdots & & l(p_i,p_j) & \\ l(p_N,p_1) & & & l(p_N,p_N) \end{pmatrix} R$$

The adjacency function $l(p_i, p_j)$ is 0 if p_j does not link to page *i*, further it is normalized to ensure that $\sum_{i=1}^{N} l(p_i, p_j) = 1$.

Google has defined a set of trusted websites, that serve as trust root for the algorithm. These websites also form the entities Google works upon. The links between websites and the previously described damping factor, form the credentials, which are combined to a ranking decision. Concerning which entities to use, the original paper proposes to use the `index.html` file of each webserver, but this seems unrealistic from a practical point of view. It is also unclear whether the starting points are identified by any means of IP address, DNS name, or a certificate. It can therefore not be reasoned about any kind of used Identity trust. Although not all facts about the algorithm in its current form are known, given the accuracy and success of Google, this work can benefit from the idea of the technique: an algorithm managing a large network of reputation, should not only take the sheer number of votes into account, but should combine it with the credibility of the voter.

3.2.3 Advogato

Advogato is a community platform for free software developers. It uses a special *group* trust metric, developed in [Lev04], to rate members of the community. *The members of this site certify each other, specifying one of three skill levels [Lev00a].* The system takes these certification *credentials* as input and uses its "Group Trust metric" as policy, which in turn decides on a trust level. Depending on the trustworthiness of a member, it receives benefits on the site. These include posting news items, comments and editing project information on the site.

Group Trust Metric The algorithm computes a "global" trust value of all nodes. Accounts and certificates are modeled as a graph, where nodes are connected via a directed edge, if a user has certified the other. Its trust root is formed by a set of "trusted accounts". Starting at these roots a shortest path search in the graph determines the distance between a "root node" and a given node *n*. Depending on the distance value, the node is associated with a capacity. The graph is then in a state similar to the system, that can be found in figure 3.6.

After these capacities are assigned, the graph defines a single source, multiple sink problem. Furthermore, capabilities are assigned to nodes, instead of edges. Therefore the graph has to be modified, by introducing a "supersink" node is added to the system. By connecting every node of the system to this new created node the problem is redefined to a "single source, single sink". As figure 3.7 illustrates, the supersink is connected with nodes, by splitting nodes into a $+$ and $-$ part. Another edge from every newly created $+$ node is added to the system with a capability of 1. The connection from the $+$ to the $-$ node is assigned the original capability of the node minus one.

The global trust value is computed by using a maximum network flow algorithm on the resulting graph, under a last constraint: If it computes flow from a $-$ to a $+$ node, there must also be flow from $-$ to the supersink. The actual algorithm used is a standard Ford-Fulkerson algorithm. This algorithm repeatedly finds an augmenting path through the residual graph, until no such path exists. After the computation of the network flow, the algorithm certifies each node, that has a connection to the supersink, based on the computed flow.

Advogato performs its kind of certification at three different levels: Apprentice, Journeyer and Master. This is actually done by running the basic trust metric three times with modified rules for creating edges.

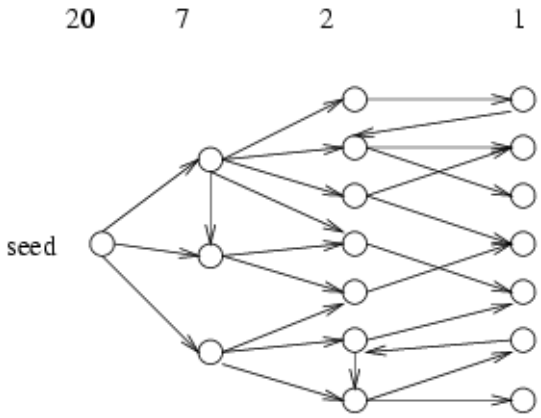


Figure 3.6: An capacity assigned graph, used to compute the Advogato trust information of a node by combining capacities of their referring nodes [Lev00b]

The Advogato system has seen much attention, as it is said to be a community of professionals, with very few not trustworthy, so called “trolling” user. The systems shows that a trust metric can be successfully derived from a maximum network flow algorithm.

3.3 Distributed Reputation Systems

This section will introduce reputation systems that are designed for operation in a distributed manner. Since these systems are based on P2P technology, the first chapter will present a short history of P2P systems. Two solutions will receive attention in the following subchapters. Beginning with an adaption of the *PageRank* algorithm, from chapter 3.2.2, a work, that is more focused on the architecture of such a system, by *Samarati et al.* will be discussed.

History of P2P P2P development started with the *Napster* system in 1999. This system allowed every node in the network, besides being a client to a service, to provide the services, acting as a server. The problem of locating information was solved with the introduction of central index servers, which where run by the corresponding company. Napster was soon followed by the *Gnutella Network* in 2000, providing a fully decentral system. P2P systems have received much attention in science over the last years. Research has been carried out about the sociological and technical aspects *etc.* . One main part of research that was put in by computer science was the question of efficiently locating information in a network, called a *distributed hash table* . These systems work similar to a common hash table. Data is identified via a key. A hash function returns the location of the corresponding data in the system. The distributed case returns the corresponding nodes on a network, the data has been mapped to. This enables every node to have a “global view” of the network. Two recent examples of this technique are the Chord system of *Robert Morris et al.* [MKKB01], introduced in the year 2001 and the *Kademlia* distributed hash table by *Maymounkov and Mazieres* [MM02]. In the year 2001 *Robert Morris et al.* proposed in [MKKB01] the *Chord* DHT, which was followed by *Kademlia* in 2002 [MM02].

3.3.1 Eigentrust

A distributed trust metric was presented by *Kamvar et al.* in [KSGM]. The system aggregates a *local trust value* s_{ij} by the ratings of transactions between the own node *i* and the peer *j*. Positive transactions increase the trust value by one, negatives decrease it by one. Therefore $s_{ij} = \sum r_{ij}$ with $r_{ij} \in (-1, 1)$ representing the ratings. The proposal goes on normalizing the values as follows:

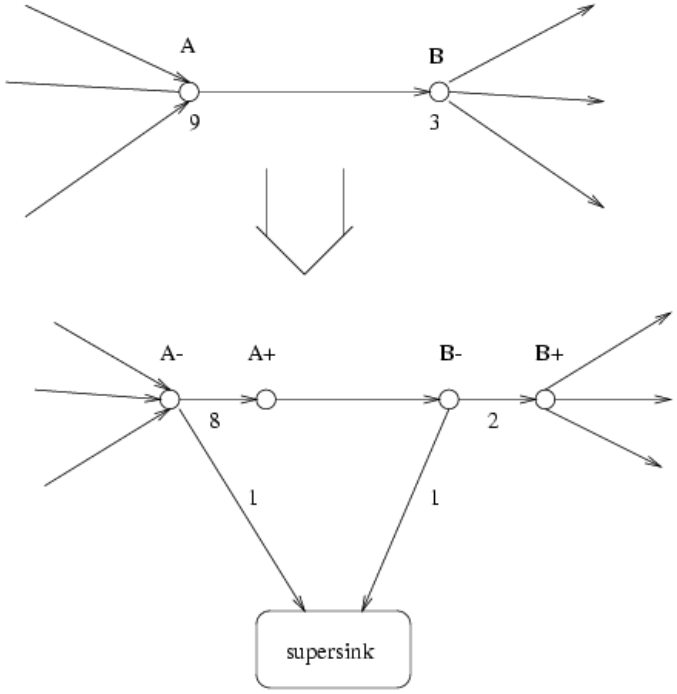


Figure 3.7: Reduction of the “single source, multiple sink problem”, to a “single source, single sink problem” [Lev00b]

$$c_{ij} = \frac{max(s_{ij}, 0)}{\sum_j max(s_{ij}, 0)} \tag{3.2}$$

Note that in the case of a zero denominator the equation is undefined. In that case the normalized trust value is replaced with a predefined trust value *p*, that comes from a predefined trust vector, that was defined before starting the system. When there is a decision based on trust to be made covering peer *k* the algorithm weightens the options of other peers depending on the trust he places in them:

$$l_{ik} = \sum_j c_{ij} c_{jk} \tag{3.3}$$

This equation could be written in matrix notation: If all c_{ij} values were be added to a global matrix - containing all local trust values of all peers - $C = [c_{ij}]$ and l_{ik} could be written as vector $\vec{l_i}$ containing all k recommendation trust values, the computation could be written as:

$$\vec{l_i} = C^T \vec{c_i} \tag{3.4}$$

Currently only the local trust values in friends and their recommendations are included. Using the matrix notation simplifies the expression of a recommendation of a friend’s friend. This would be $\vec{l_i} = (C^T)^2 \vec{c_i}$. After n for a large n the trust vector $\vec{l_i}$ will converge to the same vector for *all peers i.* $\vec{l}$ represents a global trust vector, which is the left Eigenvector of C.

Every peer can compute its own global trust value, with a as a damping factor, for including the predefined trust values:

$$l_i^{(k-1)} = (1-a)(c_{1i} l_1^{(k)} + \cdots + c_{ni} l_n^{(k)}) + a p_i \tag{3.5}$$

As a decentral computation of the trust values is possible, the proposed distributed system delegates the computation of trust values to so called *score managers.* As a P2P system is sketched, every client on the network is the score manager of another peer - his so called *daughter peer* -, computing a part of the global trust vector $\vec{l}$. To identify these peers Chord [MKKB01], a distributed hash table, that will be presented in chapter 4.1, is proposed. Besides the computation of the global trust value the peer has to maintain the local trust values of its daughter peer. To compute an appropriate trust vector, all score managers of peers that have downloaded from each others' daughter peers need to stay in contact, sharing the local trust values and computed global trust values with each other.

3.3.2 P2PRep

There have already been successful attempts to equip a P2P protocol with a trust management feature, done by *Samarati et al.* in [DdVPS03]. The System called `P2PRep`, built on top of the `Gnutella` protocol, enables peers to keep track and share with others information about the reputation of their peers. As the system relies on a P2P system as storage for reputation information it is categorized as a *distributed reputation system.* In the following the Gnutella system will be sketched, a detailed explanation can be found in [Cli01].

Gnutella The `Gnutella` network appeared in 2000. It was the first system to work in a totally decentral fashion. Previous P2P solutions like Napster relied on a central index server. The system is "open" for every client, called servent, to join. Clients are identified not by IP address, but by their associated `servent_id`, that is not assigned by the network, but is created on the fly by the client. For providing search functionality the protocol featured a new approach - the horizon. Every peer is connected with a number - commonly up to 10 - of other peers, called the "neighbours". That topology is called a mesh. If a node searches for a key, it broadcasts a `Query` request to its neighbours, which in turn will contact their neighbours. The request is equipped with a "time to live"(TTL), that is decreased when passing the search further. When the lifetime counter is zero, the search terminates. The concept is very similar to the `TTL` field of the `IP` header. For a better understanding of the taken approach, a extract from the Gnutella protocol is shown in figure 3.8 on page 26. Clients search for information in the first phase via `Query` messages and retrieve it in a second phase, from clients that have sent a *ResultSet* contained in a `QueryHit` message.

P2PRep *Samarati et al.* add two more phases to enable reputation sharing capabilities to Gnutella, *polling* and *vote evaluation. "After receiving the responses to its query, p can select a servent [...] based on the quality of the offer and its own past experience. Then, p polls its peers by broadcasting a (*`Poll`*) message requesting their option about the selected servents"* [DdVPS03]. To protect the messages from tampering the system makes use of the previously described asymmetric cryptography. `Poll` requests are equipped with a dynamically created public key. The voters will reply with a `PollReply` message containing their option along with their IP, Port and servent_id. In the enhanced version of the algorithm [1] the voter will digitally sign the data with its private key. The corresponding public key of the voter, the data and its signature, are then sent back to the polling client, encrypted with the given public key, contained in the `Poll` message. The polling algorithm will then check the validity of votes. The encryption of reply messages provides privacy and integrity features. As the contained data is signed and the involved public key is provided, integrity is even more provided. To prevent fake IPs, the algorithm uses `AreYou` messages containing the provided servent_id, sent to the IP, contained in the vote. On a successful `AreYouReply` the client can select a client based on the votes, which may be weighed using previous experience. The different phases are illustrated in figure 3.9 on page 27.

Trust Model Using the trust classification given in chapter 2.4, it is obvious, that the whole area of P2P systems needs to establish provision trust, as every peer is a service provider. Any trust related work in that topic aims ultimately to enhance the quality of service. The presented trust solution aggregates experiences with peers in a repository, as a set of triples Ψ = *(servent_id, num_plus, num_minus)*. It further aggregates measurements of voters, reflecting the number of times their recommendations met the expectations θ = *(servent_id,*

[1] For a description of the basic algorithm see [DdVPS03]

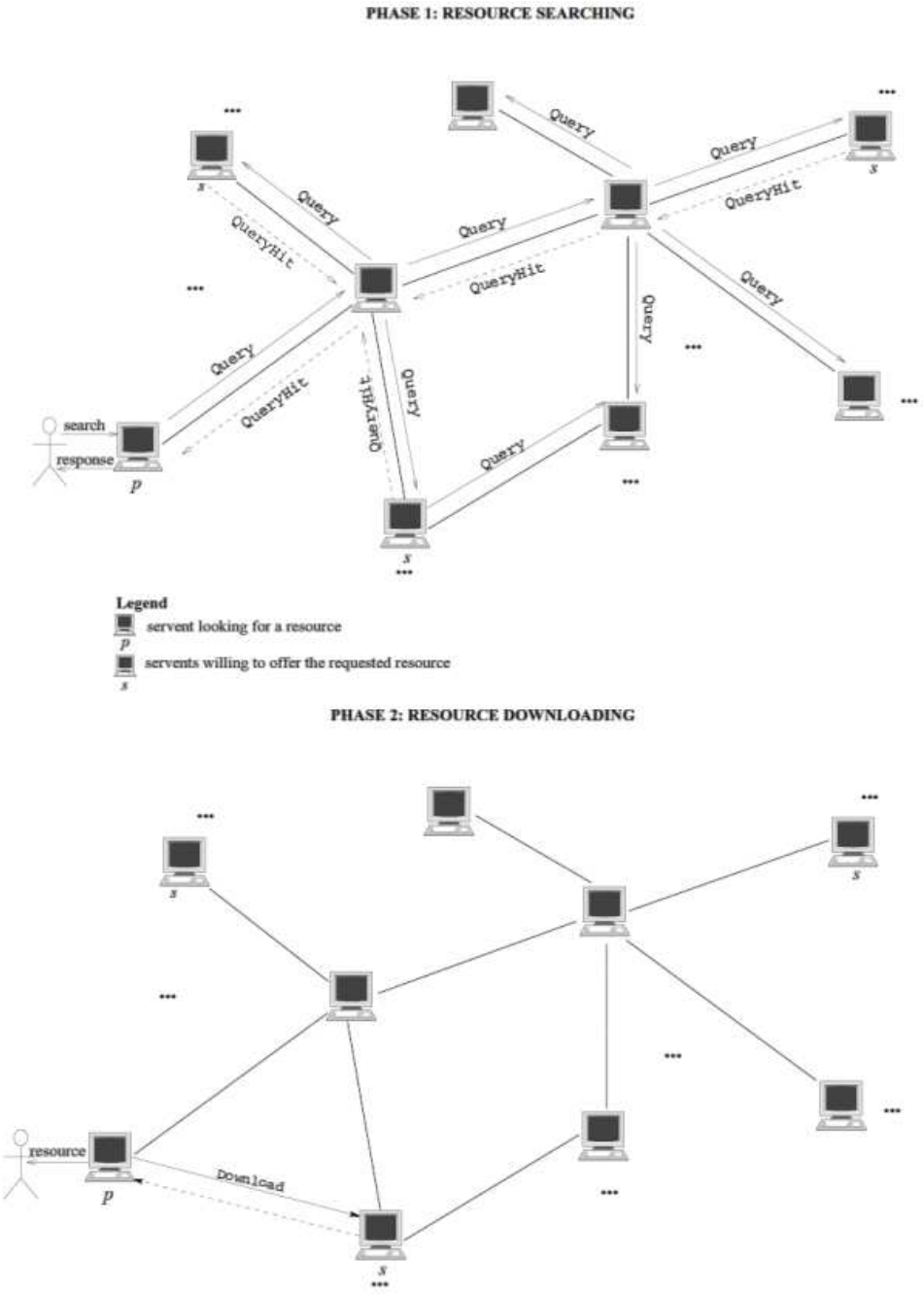


Figure 3.8: The Gnutella Protocol for searching and retrieving Information ([DdVPS03])

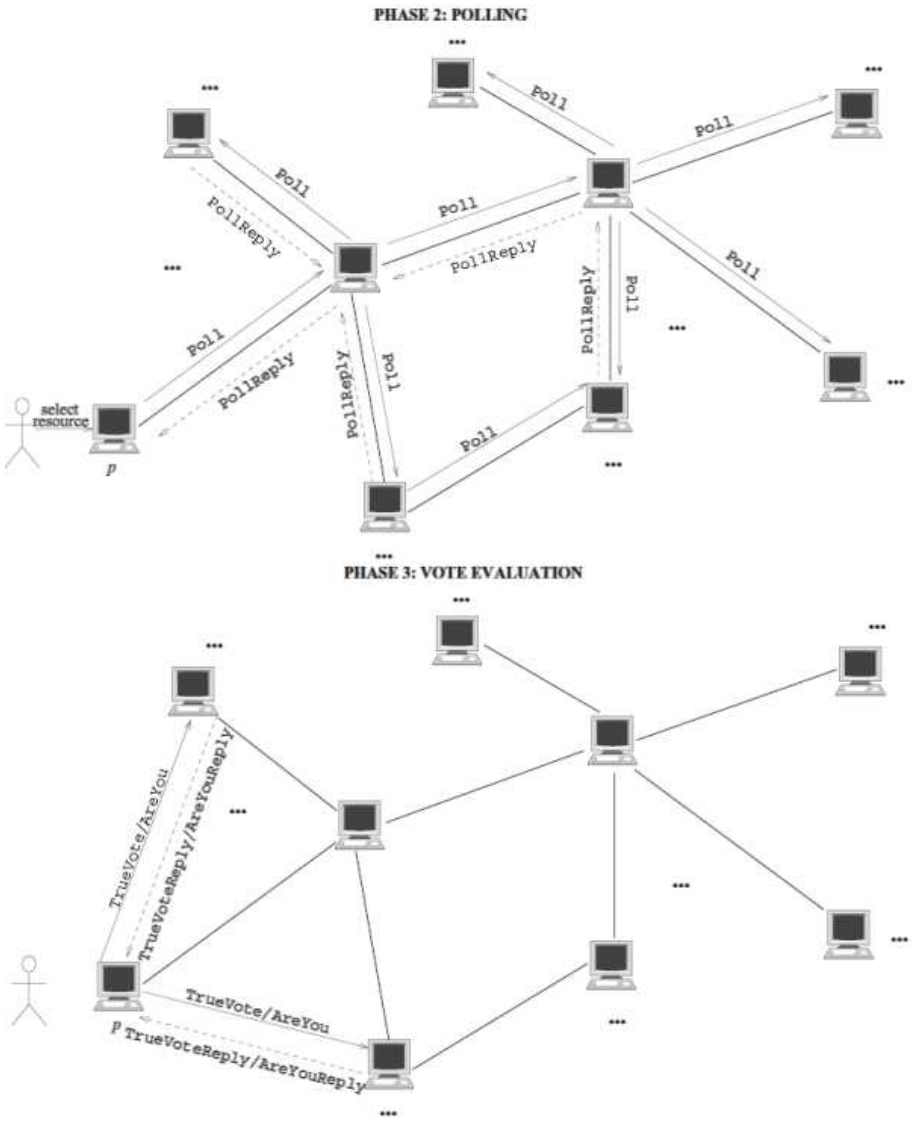


Figure 3.9: The Gnutella Protocol for searching and retrieving Information enhanced by P2PRep ([DdVPS03])

num_agree, num_disagree). Collecting information on the voters enables the system to establish a history of previous performance, not only concerning previous transaction, but also on previous reliability as voter, as a large number of *num_disagree* reveals bad performance also in this area. In other words, an entity gets a set of credits assigned that reflect its previous performance and voting actions. A policy takes these credits into account and decides, whether a relation (here: transaction) is sensible (figure 3.10). The end results of a transaction are used as input for another policy based decision, which yields an updated relation between the two peers and produces a new set of credentials.

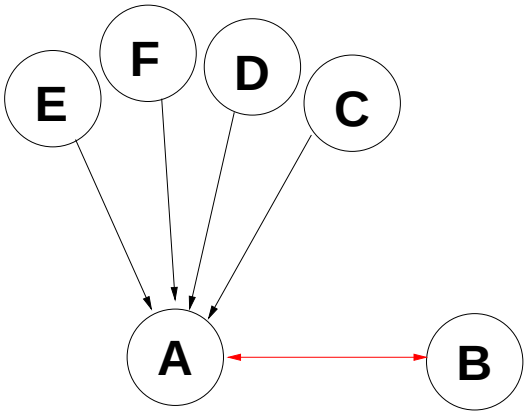


Figure 3.10: P2PRep: Aggregation of votes

Trust Metric The trust metric consists of two parts: aggregation of experience values into votes and aggregation of votes, to come to a trust decision. Votes are created using an *aggregation operator* $\phi : \Psi \rightarrow \{0, 1\}$, that can be chosen by each voter independently. The paper lists two possible definitions, a conservative approach, that defines $\phi(\Psi) = 1$ only if *num_minus = 0* otherwise it is set to zero and another more forgiving approach, settings $\phi(\Psi) = 1$ if $num\_plus - num\_minus \geq 0$. The aggregation of votes is based on the data from the θ repository. As in the previous step the operator $\phi : \theta \rightarrow \{0, 1\}$ is used. If $num\_agree - num\_disagree > k$ for a given $k > 0$ the function yields a positive result $\phi(\theta) = 1$. The paper makes a number of suggestions on computing the final trust value, including local conjunction, weighted averages, but does not go into details here.

3.3.3 Conclusion

The two presented solutions differ in complexity and trust metric. The Eigentrust algorithm roots in PageRank, used in Google, which models the trust decision as a path of variable length from a peer, representing the trust root, to another peer, with a given probability of not reaching the other peer in each step. The design of the solution implements the computation of trust at an authority other than the requester, which to the author of this work sees as a mayor design flaw. For this work aims at basing trust decisions on a similar concept as reality, it is chosen, to compute the trust decision locally. P2PRep matches this decision. It is also simpler from its design. However, the underlying architecture Gnutella allows trust decision only in a very limited scope, based on the recommendations of a horizon, which is limited, compared with the scalability of a distributed network. The P2PRep solution takes into account a credibility repository for weighting the recommendations each voter gives. However it does not require the network to compute the trust vector for peers, as they can operate on the given credibility data using any algorithm they choose.

Summing up P2PRep was a step into the right direction, but an adaption to another P2P protocol needs to be done to overcome limitations in the P2PRep system, especially the limited horizon a client is bound to and the design of the solution, which involved an enormous network traffic.

3.4 Attacks on trust metrics

Another section of research concerning trust is their resistance to certain attacks. One of the most widespread attacks, the so called “Sybil Attack” was named after the famous (alleged) multiple personality case of Sybil Dorsett. This patient, whose disease was described in a 1972 book of Flora Rheta Schreiber, suffered 16 different identities.

Sybil Attacks A Sybil attack is an attack on a system that involves a single person that creates multiple identities, with the aim of taking influence in a process of voting. John Doucer from Microsoft Research showed in [Dou02], that these simple frauds are nearly to always possible.

Most systems today, however, rely on building their trust root on a set of well known nodes. As already introduced, PageRank uses a set of websites to assure stability. Advogato and other more P2P based systems use a set of well known users. Furthermore, it is argued, that in the beginning of these systems most of its users are developers or early adopters, which decreases the probability of an malicious attack, as they are not assumed to be hostile. Besides the existence of a trust root, both previously mentioned systems feature some kind of attack resistance built into the algorithms. Googles PageRank has been resistant to most attacks to the current day. However the algorithm takes into account the number of links, that point to a page, which is a weak spot in nearly every search engine. To exploit this so called “link farms” can be found on the web. whose purpose is to create a tight web of links, providing references to each other.

Advogato The Advogato system is claimed to be more attack resistant than other reputation systems before. In [Lev00b] the author explains the robustness. It is assumed that a group of malicious nodes have entered the network. As a matter of fact the “bad” nodes will certify each other, building a closely tied mesh of computers, equal to the “good” nodes. Both networks will be connected by a small group of “confused” nodes, that initially enabled the “bad” nodes to enter. The situation is presented in figure 3.11.

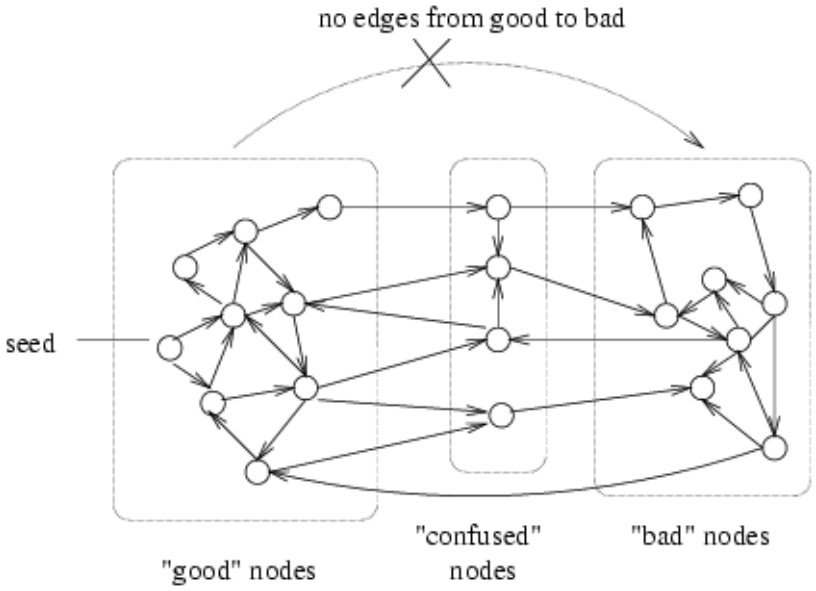


Figure 3.11: A possible attack scenario on Advogato [Lev00b]

The reason why the impact onto the network is not catastrophic is the assumption that a malicious peer will not be able to be certified by a highly trusted node on the network. As seen in chapter 3.2.3 trust declines depending on the distance from the seeding node. Given that fact the system remains in a good state, even though bad peers are able to certify each other, because this will not have a huge impact on the global trust values, seen by the system.

3.5 Conclusion

In the course of this chapter various solutions have been presented, that form - at least partially - a trust system. Table 3.1 sums them up and compares them, whether they are useful for this particular scenario.

While the solutions presented in section 3.1 are of great use providing Identity Trust, they do not fully match this models needs, as we aim at enabling Provision Trust in a distributed network. The sketched approach matches best the algorithm of the *Web of Trust* from chapter 3.1.4. Both solutions provide a decentral public measurement of trust, using features from cryptography, to protect their information. These solutions, however, differ significantly.

The only public trust information given by the Web of Trust is a signature, expressing the confidence of the signer, that the signed key is *really* the key of that person. The possibility to rate a peer is both local only and targets only the peers reliability in verifying the relation between keys and owner. The hereby presented solution publishes information expressing if the peer is trusted or *not*. This difference is critical to note, as the web of trust does not allow to express distrust, even in its scope. A user can choose *not* to sign a key if he does not trust in the key - owner binding, but can not publish this information. This lies in the architecture of the system. While this solution creates a public available repository of *all* available information, the Web of Trust centers around creating a path from a clients own key, to another key, by combining signatures of already known clients, with the level of trust he puts in them. Beeing restricted to information that is only reachable using this path, the client is not able to use all available information in the system. Especially P2P systems are considered, to be too huge to build a sufficient trust path to other users, especially since these are limited in size. Therefore this approach bases its trust decision on the majority of recommendations, but includes also the ability to weigth every vote, by a local “credibility” factor.

As a last point the Web of Trust requires a static key infrastructure, that is not flexible enough for the needs of a P2P system. Public keys need to be published at a central instance, which destroys the idea of a real P2P system. This solution uses a self signed PKI, in which every client can request the public key of another client from this client itself, which reflects truly the P2P idea. More information on this solution can be found in chapter 5.

Furthermore there have been centralized systems, that are able to compute a global trust value, upon receival of new input, or upon periodic execution of an algorithm. These system feature elaborate trust metrics, ranging from vector geometry to network flow algorithms. While serving as a good example for reputation, or trust systems in general, the architecture is quite different to the decentral structure, we want to research on. All systems require clients to fulfill some constraints, that enable them to access the system, while we focus on a “open” approach. Similar to the previously mentioned cryptography, they serve as the foundation of our work.

In chapter 3.3 existing distributed solutions received attention. For the reason stated in chapter 3.3.3 P2PRep clearly meets the needs the requirements of our scenario best. Recent development in P2P show, however, that the chosen basis, the Gnutella protocol suffers some weaknesses. Firstly, the limited horizon does not provide a global view on the network, while still requiring enormous traffic. Secondly the search algorithm is in no way as efficient as algorithms used for distributed hash tables, that manage to provide search operations in logarithmic time. Thirdly is the enormous message overhead on the network. Every client that needs to establish a trust relation needs to poll its horizon for votes. Even if another client starts the exact same poll, messages are by no means cached.

For that reasons we will base our work on the fundamentals researched in its surroundings. We will, however, change the underlying protocol to a P2P Protocol based on a distributed hash table. This will be chosen in the next chapter.

	Trust Model	Distributed	Global Network View	Trust Computation	Cryptography
X.509	Identity Trust	No. System uses one central trust root	Yes. Central Server	Client side	Yes
Web of Trust	Identity Trust	No. System uses one central server architecture	Yes. Central Server	Client Side	Yes
Blaze	Access Trust	System uses central server architecture, supports delegation	Yes. Central Server	Client Side	Yes
PageRank	Provision Trust	Central approach	Yes. Central Server	Server Side	No
Advogato	Provision Trust	Central approach	Yes. Central Server	Server Side	No
Eigentrust	Provision Trust	Yes. P2P Approach	Yes. P2P system	Not local, delegated to other peers	No
P2PRep	Provision Trust	Yes. P2P Approach	No. System relies on Gnutella, that provides only limited horizon	Client Side	Yes
This Solution	Provision Trust	Yes. P2P Approach	Yes. Distributed Hash Table based, provides all votes available	Client Side	Yes

Normal Text	Met Requirement
Grey Text	Unmet Requirement

Table 3.1: Comparison of all solutions

4 Choice of a Peer to Peer Model

As seen in the previous chapters there are a variety of Reputation Systems and Trust Metrics already in use today. However, this thesis covers a distributed reputation system incorporated into an existing P2P system, because these system feature the following benefits:

- Decentralised: Not a single server; the whole network provides information
- Broad userbase: The Kademlia DHT is said to currently feature up to 5 Million users at a time
- Frequent transactions between strangers: P2P is an n to n relation, central systems have a n to 1 ratio
- Speed: As information is distributed, load is balanced across the network

Copyright As those systems are abused for any kind of thievery the author wants to once more clearly separate his work from these frauds. This work wants in no kind support the violation of copyrigth law. In the meantime however even commercial interest in P2P has been awakening. This seems to be rooted in the fact, that distributed network solves a problem that vendors of digital media over the Internet face. A typical pop song encoded with a compressed format consumes roughly 3 Megabytes of space. Classic centralised vending platforms like the *iTunes Music Store* prove that they can handle the distribution of digital music. The iTunes Japan store sold 4 Million tracks in four days, which would be based on the previous assumption 12 Terabytes of Data. But with the advent of better Internet connections and video codecs, digital media distribution is no longer limited to music. Video data, which takes from 700 Megabytes of space for acceptable quality, setting nearly no limit on space consume for high end data.

Providing this load of data to consumers is a challenging task, as bandwidth costs explode. This can be solved by employing P2P technologies, as already done by *www.peerimpact.com* . This solution enables customers to download data directly from other customers. The only need for a central server are the billing part of the transaction. Customers providing their data for download get credits for providing their bandwidth. Thus a cheap bandwidth saving opportunity is employed which offers even more benefits, like the reliability of a distributed network.

Copyright violations, however may prove that the feeling of anonymity is on these systems imminent. Basically, this is the case, because most P2P implementations actively hide the identity of the user. In conjunction with the large userbase of P2P techniques, providing a lot of unknown users, serves as a basis for the spread of abuses. A first fraud, that is cited in combination with P2P and security problems is the `Gnutella.VBS` worm [KSGM]. The worm appeared in 2000 and replicated itself upon execution, by copying itself into the Gnutella shared folder under different names. It also modified the configuration of the program, to allow its distribution. As users need to actively click on the worm to become infected, one could argue, that the impact of this worm could not be that large. However, worms that spread via emails need also to be activated by user interaction and in spite of this manual invocation they have been very successful.

4.1 Chord

In 2001 the Chord distributed hash table was developed at the MIT Laboratory for Computer Science. A distributed hash table works equal to its local counterpart. It maps keys to values, using a given hash function. However in a local case, information may be a pointer to a memory location, while this kind produces pointers to nodes in a network.

Network Topology The architecture of chord is a ring, clients joining the network will be included by inserting them between their corresponding neighbours. See also figure 4.1. Basically a node only needs to know its successor, to be able to pass a query on the network to its successor. For efficiency reasons clients in the chord ring have routing information, commonly referred as "finger table", containing a list of nodes. *The i^{th} entry in the table at node n contains the identity of the first node, s, that succeeds n by at least 2^{i-1} on the identifier circle [...] [MKKB01].* By using this technique chord *resolves all lockups via O(log(n)) messages [MKKB01].*

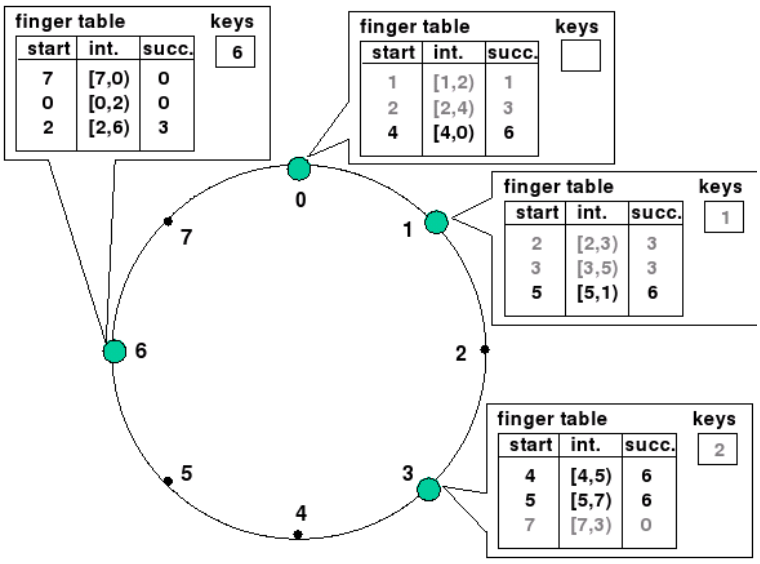


Figure 4.1: A sample Chord Network, equipped with nodes and their routing tables([MKKB01])

Simultaneous Node Joins The described architecture works fairly well when used in a sensible manner. However when scaling the system to the size of the Internet, the weakness of the system can be identified as the need for a node, to know its successor. Nodes joining the system, need to make other nodes aware of their presence. If multiple nodes join simultaneously the network, it is possible that the mechanism fails and the ring is split up into two or more fractions, as described in the original paper. Normally, this problem is caught by a stabilisation protocol that is needed, to keep the ring consistent. However, in the worst case the system ends up with two rings, that both appear consistent to the stabilizing protocol. In [MKKB01] it is noted, that there is currently no way to detect this kind of failure.

4.2 Technical Introduction to Kademlia

Kademlia is another peer-to-peer distributed hash table (DHT), developed at the New York University in 2002. While providing the features of a DHT, it adds some unique features which make it fast and scalable. Key benefits are:

- Small protocol, not requiring an extra configuration protocol
- Speed: $O(n) = log(n) + c$ for a system with *n* nodes [MM02]
- Efficient Caching Mechanism
- Open Source Implementation

Before explaining the functionality of the Distributed Hash Table, it is substansial to understand the basics of the network. This subchapter will start with the employed network topology, and proceed with the distance metric, most operations in the network are based upon. After revisiting the basic protocol, it will be possible to understand the used routing system and it's benefits.

4.2.1 Network topology

While chord's network topology is a ring, Kademlia is modeled as a binary tree, with each leaf representing a node. Figure 4.2 illustrates this. The difference between the two solutions is obvious: A ring requires every element to know at least its successor. To ensure this assertion, a stabilisation protocol is required, as similar joins to the network could destroy the topology of the network. A binary tree solution needs none of this special treatment. Similar to Chord, each node in the tree is identified by a unique identifier, which happens to be a 160 bit key.

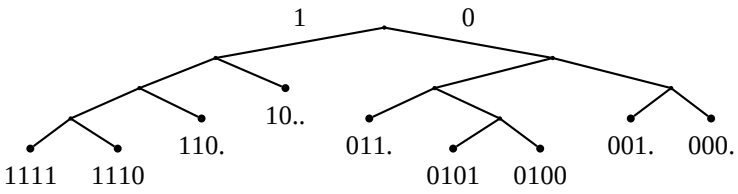


Figure 4.2: The Kademlia binary tree with 4 bit IDs: Leaves in the tree are nodes, dots in the IDs represent hidden subtrees

4.2.2 Distance Metric

The paper defines a special distance metric for Kademlia. Distance is computed by using a bit wise `XOR` of a nodes ID with a target ID. As all distances are computed bitwise, numbers, expressing distances will in the following be written in binary notation. An illustrated example, as in figure 4.3 will help for a better understanding:

a)

xor	0	1
0	0	1
1	1	0

b)

```
  011
xor 110
= 101
```

c)

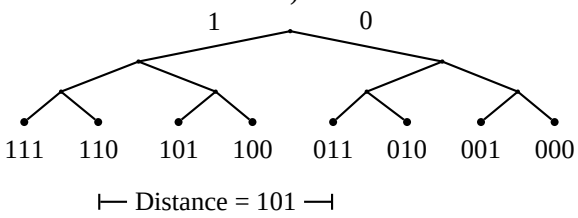


Figure 4.3: The Kademlia Distance Metric for 3 bit node IDs: Truth Table of XOR a), sample distance computation b), illustration of b) in c)

The Distance between two nodes, for example 110 and 011, is 101, as seen in the a) and b) part of the figure. An interpretation of this number is illustrated in the c) part of the image. This distance notion is not the traditional notion of distance, but maps implicitly to distances in the binary tree. Huge distances mean very distant subtrees, small distances represent "close" subtrees.

4.2.3 Protocol

Kademlias protocol is designed to have a small footprint. It relies on 4 primitives.

- `Ping:` This command checks for a hosts existence, similar to ICMP requests.
- `Store:` A request to publish information is sent to a given node, including the key, which identifies the information and the data itself. The target node will then check whether a given key should really be stored at its location. This is done by employing the already mentioned distance metric.
- `Find_Node:` The basic routing primitive of Kademlia. It is used for location other peers in the tree. It employs a recursive algorithm that is described in a following paragraph.

- `Find_Value:` A request to locate a given key in the node ID space. It works in the same manner, as the routing primitive.

It has already been stated, that no specific stabilisation protocol is needed. Further the efficiency of the system is increased, as ever message of the protocol will be taken into account for routing functionality.

4.2.4 Routing

The protocol primitives `Find_Node` and `Find_Value` both are based on the routing subsystem. Routing information is held in a data structure called k Buckets, where k determines the size of the buckets. *"For each $0 \leq i < 160$, every node keeps a list of [...] nodes of distance between 2^i and 2^{i+1} from itself"*[MM02]. The meanin of distance in the context of Kademlia has been introduced previously in the paragraph "Distance Metric". The range of i is chosen, because of the 160 bit identifiers, which allow 159 subtrees, that do not contain the node. The size of the buckets, k is suggested to be set to 20, which represents the *maximum* size, a bucket can grow to, smaller, or even empty buckets are valid. If a represents a node and ID is the nodes own identifier, it can be expressed as:

$$\forall_{a \in k_i} : 2^i \leq a \oplus ID \leq 2^{i+1}$$

A graphical representation of a tree with 4 bit IDs and $k = 2$ is presented in figure 4.4 accompanied by table 4.1.

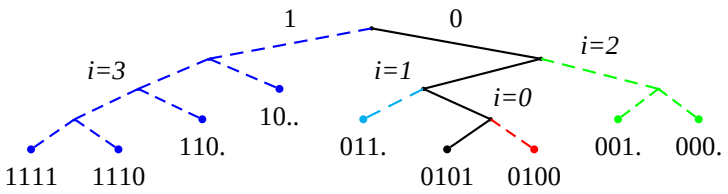


Figure 4.4: The Kademlia routing tables illustrated. Dashed subtrees represent the 4 Buckets available in the tree.

Bucket	**Node List**	**Bucket**	**Node List**
i = 0	0100	*i = 2*	0010
			0011
i = 1	0110	*i = 3*	1111
	0111		1000

Table 4.1: A sample Kademlia routing table, for 4 bit node IDs and a bucket size $k = 2$

These buckets are filled with nodes, which are set up and kept up to date by using the information network traffic provides. When nodes exchange messages, e.g. for searching, the contacted node can not only return information to the requester, but also add him to the corresponding bucket in its routing table. If the client is already contained, it will set the client alive, marking it with a "last time seen" timestamp.

Because nodes are stored in buckets, the routing algorithm can choose the "optimal" path along nodes regarding speed or node failures by contacting a subset $\alpha < k$ of nodes and in turn contacting the fastest replying node. So the routing system is failure redundant.

Of course the k factor limits the size of the bucket. When the bucket is full, nodes could no longer be added to the bucket. It would be convenient to add the contact immediately to a bucket, dropping another client. Research however has shown, that the probability for a client going offline decreases, the longer he stays online. So the routing algorithm will prefer already known clients over unknown clients and *not* add it to the bucket. This technique makes the algorithm resistant to a number of Denial of Service attacks, as attacking clients can not easily fill the targets routing tables with bogus, or their own, entries.

Buckets are further periodically checked for offline nodes by sending `Ping` commands to nodes, that have not been set alive in a while. If these do not reply, they are removed from the pool and considered offline.

After discussing the involved data structures the routing algorithm will receive attention.

4.2.5 Finding Nodes in the DHT

The implementation of `Find_Node` is a recursive algorithm. The Figures on the following page present a graphical illustration of the process.

1. The searching client with ID 001 looks up the target node 111, in its routing tables. If it is found the algorithm terminates and returns an IP address. Otherwise it picks the "closest" node to the target node, from its routing tables - in this example the client with the smallest distance to 111 will be 101 - and executes step 2.
2. The chosen client is contacted by 001 (Figure 4.5). Again, "close" means that the bitwise xor of the target node's and chosen node's ID is minimal. Upon receiving the query, the chosen node looks up 111 in its routing tables and similar to step *1* returns a $k-$sized set of closest nodes, either containing the searched node, or just closer nodes, e.g. a node 110.
3. If the queried node returns the target, the algorithm succeeds (Figure 4.7), if the result yields no closer nodes, that the already known the algorithm terminates, reporting a failure. Otherwise a new target node is picked from the set of results and step *2* is repeated (Figure 4.6).

During each iteration of the routing algorithm, client 001 might be added or set as alive in the corresponding *k* Bucket. This information is, as described earlier, used to enforce the usage of long living clients in the routing tables. Furthermore this is an example on how closely configuration messages are integrated into the core protocol primitives.

4.2.6 Publishing information

Publishing information on the network is the task of adding the data to the distributed hash table. It involves identifying the node where the information needs to be stored, by sending a `Find_Value` command and uploading it using a `Store` command.

Keys Kademlia addresses both data and nodes as 160 bit keys. The function of the Distributed Hash Table that maps the keys of data to nodes on the network is indeed very simple: The k closest nodes to a given *data* key will store the values of the keys. So the binary tree structure of the node network is resembled in a parallel data tree. A simplified illustration can be found in figure 4.8. In this example data and node keys are only 3 bit wide. The table shown in that figure consists of data keys, computed from the file and the corresponding node, where data would be mapped in the right hand side binary tree. For example, the key of file B is 001. It is mapped to node 000, as the distance between key and node is 001, while the distance between key and node 010 is 011.

Publishing Adding information to the network employs 3 steps. First, a corresponding key to the value is computed, using a well known hash function. The paper suggests the SHA-1 hash function, but this is implementation specific. The second step is retrieving the *k* closest nodes to the returned key by searching for the closest nodes to this key on the network. This is exactly the basic routing algorithm, that has been discussed in the previous chapter. When the set of the closest nodes is found, a `Store` command is sent to these *k* nodes, requesting them to publish the information together with the key at their position.

By storing at *k* nodes, an efficient cache is installed, because nodes searching for information will have a list of the k closest nodes. By contacting a subset α of k, the fastest replying node can be contacted. To avoid loosing information, by node failures, or nodes closer to that key joining the network the key must be republished in a given interval, usually one hour.

4.2.7 Retrieving Information in the DHT

Retrieving information is a two phased task, similar to a normal hash table. First, a key has to be found, second the information has to be retrieved.

Finding Information As the user can not compute the key of a set of data, he does not own, several ways to publish those keys exist. It is common practice to publish a key on a given website. For example most Linux distributions use this technique to publish pointers to their free products on their homepage. Independently from other networks, the information can be published on the network itself. By publishing a hash of common search strings or the filename along with a pointer to the *real* hash key, an easy way to locate the desired key can be established.

Retrieval For finding a value corresponding to a known key, the algorithm resembles the routing algoritm. A lookup of the closest node to the given key via a `Find_Value` command is performed. In the recursive step, the recursion breaks if hitting a node that stores the required value. In that case, the node will return the value. Otherwise it returns a set of closer nodes and the recursion continues. So the only difference between locating a node and retrieving information is the returned value, instead of an IP. Besides the routing configuration information, this search spreads across the network. The design features also a caching mechanism described in the next paragraph.

4.2.8 Caching

Kademlia caches ⟨key,value⟩ pairs across its network. This is done after information lookup: a node, that has successfully completed a query will store the result at the last node, that did not return the information. By installing such a cache it shortens the path of its query by one routing message. The probability of other nodes hitting such a cache is high, because the xor distance metric is unidirectional. This means that the closer a peer is to its target, the more likely it is to hit a path another client already used. For a proof, see the original paper [MM02]. Therefore, if the key is popular this caching technique enables fast lockups, as it “balances” the load. A side effect is that even after multiple node failures the information is likely to be available in the network. The information itself is already stored not only on one host but is distributed to the *k* “closest” hosts as seen in the section Publishing in subchapter 4.2.6.

4.3 Conclusion

The need for a stabilizing protocol, that need to maintain the integrity of the Chord ring, explains the choice of the Kademlia solution, which is described to be self configuring and maintains its stability via the normal search operations.

From a clear practical aspect, the decision is based on the fact, that Kademlia is in widespread use by the *eMule*[1] client. It is the second implementation of the system, besides the reference implementation. There is also at least one more public available implementation in the *Arzeurus*[2] client. All of these implementations are open source, which enables modifications to the code base in an easy manner. The available implementations have also proven, the chosen solution scales up to 5 Million users, according to a developer of the eMule client.

Table 4.2 sums up the reasons, that lead to the choice of the Kademlia system.

[1] www.emule-project.org
[2] www.arzeurus.org

	Chord	Kademlia
Implementations	1	3
Implementation Maintained	No	Yes
Protocol Overhead	Stabilisation Protocol	None
Search Operations	log n	log n

Normal Text	Met Requirement
Grey Text	Unmet Requirement

Table 4.2: The Chord and Kademlia distributed hash tables compared

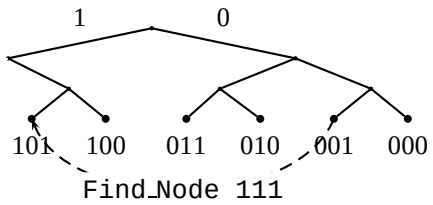


Figure 4.5: Searching for node 111: Node is not contained in the routing tables of 001, contacting a known node 101 close to the target

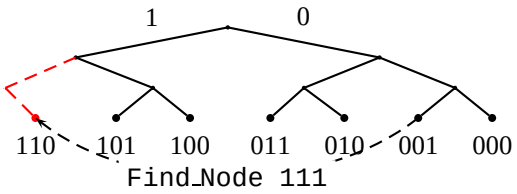


Figure 4.6: Searching for node 111: Contacting node 110, which is closer to the target, after updating routing tables with results from 101 (dashed)

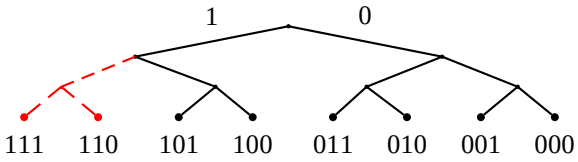


Figure 4.7: Searching for node 111: Found node and updated routing tables with results from 110 (dashed)

	Key	**Node**
File A	110	111
File B	001	000
File C	111	111

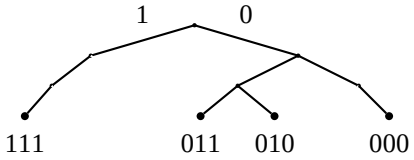


Figure 4.8: Mapping data keys to nodes

5 Model of a Trust-Aware Solution

A vital part of a network of potentially anonymous[1] users is not only the availability of information, but also their quality and integrity. To reach that goal, this chapter proposes a reputation system, based on sharing information about users, among clients. The subsequent subchapters will introduce the design of policies and a proposal for an integration into the Kademlia network, providing a secure facility for reputation sharing. The next chapter 6 will introduce the details of the reference implementation of the system.

5.1 Sketch of Design

The proposed solution is designed to be a distributed reputation system. It works, as described in chapter 2.5 by enabling users to express their experiences with other users in a way that every other user could base his decision on the recommendations of other users. In figure 5.1 on page 40 this procedure is divided into three different phases. In the first phase user A receives a set of votes about a third peer B, from another peer C. A was able to locate this information, as the distributed hash table provides a *mapping function*, see chapter 5.4, that takes a peer's identifier as input and returns another peer's id, that is chosen, to collect all votes concerning the given peer. This solution is based on the Kademlia distributed hash table, therefore the information is distributed to multiple peers, to improve reliability and balance the load between nodes.

Peer A has in the second phase a number of votes, consisting of meta data and a binary identifier expressing whether the voter recommends a transaction with that peer, or not. A description of a vote's components can be found in chapter 5.3. The decision of peer A can be based on these votes. The received data is a *collection* of votes, so the peer has to combine all the received data, to form its decision. Other systems already return a global trust value, already combined. This solution, however, allows the peer to form its decision independently.

A may have no previous experience with peer B, but it might have had interaction with some of the peers, e.g. D, that expressed their recommendation. If so, it should weight the votes according to the voters previous performance. The Policy section in the next subchapter will introduce this decision phase.

If A chooses to interact with peer B, it is able to rate this transaction after it has finished. By rating the authenticity of the exchanged information, separating an attempt to provide malware from a valid exchange of information, the peer distinguishes between a positive and a negative outcome. This information has to be published into the network, by assembling a vote and publishing it to net node, the set of votes was received from. The details of the publishing are found in 5.5.

In a final step, A is capable of remembering, how helpful the recommendations of the voters have been. If the outcome matched the recommendations of a voter, an agree value could be increased, otherwise a disagree value will be updated. More on these *experience repositories* will follow in the next subchapter under "Credentials".

5.2 Expressing Trust

In chapter 2 it became evident that there are different ways to establish trust, based on sets of credentials and polices. The basic components used in this work will receive attention first.

[1] Here: not known among each other

Trust Classes Every Trust System is a social network with entities and the relationships between them. In this model, entities are agents that behave on the actions of a user. It is the goal of this work to enable them to base their actions on a trust decision. Chapter 2.4 defined components of trust and also a classification of trust. In the course of chapter 3 it became evident, that a trust system for P2P ultimate deals with *provision trust*. The assumption that every class of trust is based on *identity trust* is also central to this work, as entities need to be able to identify each other.

Credentials To have a history of an entity's prior performance, another entity that interacts with it, has to measure the outcome of this interaction and store it. By publishing this information, a decentralized reputation system, based on the experiences of its users, can be built, providing trust relations. In figure 5.1 this is illustrated. An entity A can retrieve credentials, of node B, published at node C. Depending on the credentials it can interact with B and assign, based on the interaction a new set of credentials to B, that need to be republished to C. When analyzing the workflow of such an entity, two actions that serve as a sensible basis for credentials can be identified. Clients actively search and retrieve information. A first credential is therefore the authenticity of the provided information. In other words: "is the provided data of client X relay what it was said, that it would be? Is it infected with viri?". This credential will be called *reputation as download source.* The second important measurement lies in the nature of the reputation system itself. User's experiences with other clients should be used, but what happens if they are dishonest? To provide an appropriate measurement for the voting information a client provides, the second credential will be called *reputation as voter.*

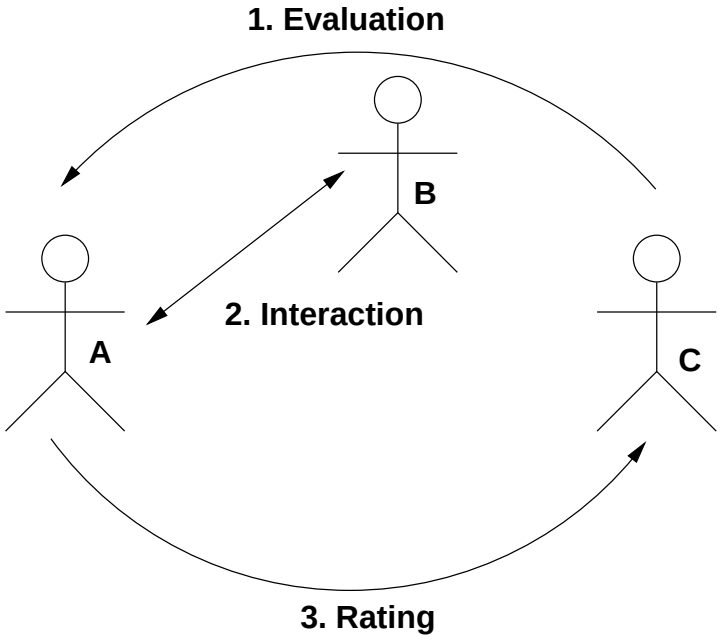


Figure 5.1: The workflow of a Reputation System. Credentials of user B are retrieved from a node C to A. On this basis a trust decision is made. New assigned Credentials, based on the interaction will get published back to C.

Aggregation of Credentials To collect a set of credentials a number of repositories is needed to record both good and bad performance of a given node. In the previous paragraph, two measurements for client's performance were identified: *reputation as download source* and *reputation as voter.* In chapter 3.5 it has been decided to base this trust solution closely on the P2PRep solution by *Samarati et al.* presented in subchapter 3.3.2. Therefore, the same nomenclature will be used. So the *reputation as download source* repository is a set of triples Ψ = *(servent_id, num_plus, num_minus)*, while the *reputation as voter* repository will be called θ = *(servent_id, num_agree, num_disagree).*

Policy To form a trust decision the the collected information needs to be evaluated. The evaluation is done by the policy of the system. While primary research will focus on developing trust by reputation, the system is flexible enough to cover a broader spectrum of of trust decisions. In the following, a policy for trust management based on reputation is presented. In the comclusion another possible policy, which establishes trust by the usage of a hierarchical structure and adapted credentials, will be sketched.

To keep consistent with the P2PRep approach, the policy is using the *aggregation operator* $\phi : \Psi \rightarrow \{0, 1\}$ that was already used in the context of subchapter 3.3.2. A forgiving strategy will be selected, that yields a positive recommendation, even if there has previously been "bad" behaviour, if the number of positive experiences with a peer is greater or equal to the number of negative ones. If there is no previous interaction, a positive result is yielded. The value is a public measurement of trust of a voter *i* into a client *j* , which will be call the *global trust value* v_{ij} in this work. When publishing this value along with metadata described in subchapter 5.3, it will simply be referred as "vote". The publisher *i* will be called *voter*, *j* will be called subject.

$$v_{ij} = \phi(\Psi_j) = \begin{cases} 1: \ num\_plus - num\_minus \geq 0 \\ 0 \end{cases} \tag{5.1}$$

For consistency reasons, the computation of the "reputation as voter", ϕ is defined as $\phi : \theta \rightarrow \{0, 1\}$. The number of times the vote matched the experience of the client must be greater or equal to the number of times it did not match the clients experience. The latter is a private value, that measures, whether the votes of a client *j* can be trusted; its credibility c_j.

$$c_j = \phi(\theta_j) = \begin{cases} 1: \ num\_agree - num\_disagree \geq 0 \\ 0 \end{cases} \tag{5.2}$$

Given these definitions, the global trust relations are limited to a binary attribute. This is expandable and will not limit future work. For the current scope however it is assumed to be sufficient for most use cases.

It should also be noted, that the aggregation operator is *client dependent.* A client may use a forgiving strategy, as presented in this subchapter. Other clients could be more strict and vote negative after the first negative experience (*num_negative* > 0).

Before describing the publishing of votes in subchapter 5.5, the terminiology needs to be explained. Since this solution is designed to be a distributed network and these values will be stored on the network, it is convenient to define a name for a peer that holds the information about peer *j*. This node will be called the *storage peer* S_j of peer *j*.

Aggregation of locally Credentials The aggregation of θ and Ψ will be explained in this chapter. Aggregation of credentials happens local on every client. The credential *reputation as download source* will be republished, if the value changed. Commonly the better a reputation system is integrated into an application, the more information can be collected without any user interaction. Automatic collection of reputation information could already take place at the block level: existing Kademlia clients use hashes to detect corrupted parts received from the network, so the client can record and use these exceptions to adjust the trust level of that peer. Furthermore after a successful transaction the file type could be checked automatically using a similar approach as in the UNIX `file` command. In this solution it is done by the user, who needs to rate peers directly on the basis of a completed download. If a peer is positively rated, the credibility of the voters θ could be adjusted automatically, by adjusting the *num_agree* and *num_disagree* values.

5.3 Layout of Votes

The two most obvious elements of votes to enable reliable and secure operation is a Tuple consisting of the information needed for reputation sharing and a digital signed hash of this information. By distributing the information in this form, it is certain that - given the fact that the signature check succeeded - the data has not been tampered either by a malicious peer or the storage peer itself.

Furthermore the ID of the subject of the vote needs to be added to the meta information. As votes may change over time, it is absolutely necessary to add a timestamp, which prevents malicious peers from sending outdated information. As the outdated vote would seem valid, there would be no possibility to detect this fraud. These frauds are called “replay attacks”. Adding the IP address or similar data for evaluation votes, as done in previous work is depreciated by the usage of cryptographic methods. More work on security can be found in section 6.4. In summary, up a vote is proposed like this:

$a_j = v_j + ID_{voter} + ID_j + \texttt{time()}$

$vote = a_j + encrypt(sha1(a_j), private\_key)$

a is a temporary placeholder. The basics of digital signing can be found in chapter 3.1. Note that explicit knowledge of the voter’s public key is required with this definition. It is not possible for a peer to check the validity if he does not have the public key of the voter. If the public key were included in the vote, it would open the door for a large scale fraud by the node storing the vote, as it could provide an infinite set of votes, all signed with bogus keys. This fundamental data structure enables us to store votes, even at untrusted places, as the data can not be tampered, or replayed. The only fraud that could happen, is that the data gets dropped by the storage. As Kademlia caches the information on *k* notes, this is unlikely.

5.4 Mapping Function

Kademlia maps (key,value) pairs to the closest node. A reputation system built on that basis should also store votes in a similar fashion at a central place. To accomplish this a function is needed, that maps given IDs of clients to a key. This is needed to identify the *storage peer* s_j that handles the reputation information of peer *j*. A constraint is that this function never maps $f(x) = x$, or following the nomenclature of that work $s_j = j$. This means that no client should ever be able to manage votes about himself.

The original Kademlia paper suggests the SHA-1 hash for creating keys of data. This choice is driven by the need to identify different chunks of data uniquely and needs to be 160 bit wide. This solution could easily rely on a much simpler function, as a 160 bit key is already present and only a direct mapping of data needs to be prevented. The `not` function ($\neg$) will be used bitwise on the subject’s ID. This function does not only prevent direct mapping between ID and storage peer, but also maps to the “most distant” node on the network, in the xor distance metric. To prove this, look at the definition of xor, with $\oplus$ meaning xor and $|$ meaning not and:

$x \oplus y = (x \mid (x \mid y)) \mid (x \mid (y \mid x))$

the definition of *not and* is

$x \mid y = \overline{(x \wedge y)}$

and then compute the distance between x and $\neg x$

$$
\begin{aligned}
& x \oplus \neg x = \underline{(x \mid (x \mid \neg x))} \mid \underline{(\neg x \mid (x \mid \neg x))} \\
& = (x \mid \overline{(x \wedge \neg x)}) \mid (\neg x \mid \overline{(x \wedge \neg x)}) \\
& = \underline{(x \mid 1)} \mid \underline{(\neg x \mid 1)} \\
& = \overline{(x \wedge 1)} \mid \overline{(\neg x \wedge 1)} \\
& = \overline{\neg x | x} \\
& = \overline{(\neg x \wedge x)} \\
& = 1
\end{aligned}
$$

Given 160 bit identifiers, the distance would be $2^{161} - 1$

Since Kademlia identifiers are free to choose, this leaves room for malicious users attacking the system with clients managing each others reputations, dropping negative votes. But the fact that the system publishes each information k times makes this increasingly difficult. A clique detection algorithm together with an add on to the reputation system, described in Section 6.4 should minimise that impact even more. Solving this Kademlia-wide problem is outside the scope of this work.

5.5 Publishing a Vote

The most visible change by adding a reputation system is the ability to "rate" a peer. By presenting the user the possibility to express his satisfaction about other users, judging if the provided information in terms of viri or authenticity, the system can update Ψ accordingly, effectively updating its local trust relation to that user. To update the global information the following steps are required.

To publish a vote, concerning a node ID 0000, the voter 0100 needs to prepare a vote, as described in chapter 5.3. Then it computes the storage peer s_{0000}, that handles votes for 0000, as described in the previous chapter. Similar to the algorithm described in 4.2.5 the set of k closest nodes to s_{0000} are computed via a `Find_Node` command. These nodes will receive a `Store` command containing the vote. The recipient of this message will store the vote in a basket associated with 0000 (Figure 5.2). Further action is not required from the recipient. Error checking or building a mean of all trust values would be insecure and is therefore a duty of the client. Other clients searching for votes can now retrieve this updated basket.

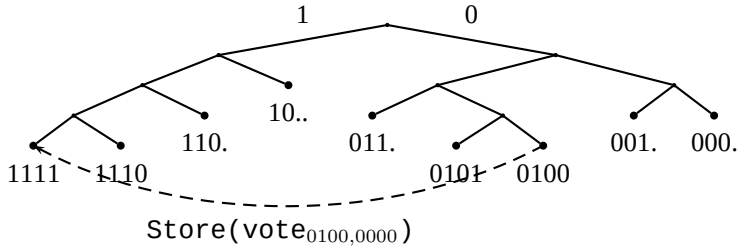


Figure 5.2: Node 1111 is contacted by node 0100, to store a vote on 0000 in its associated vote basket

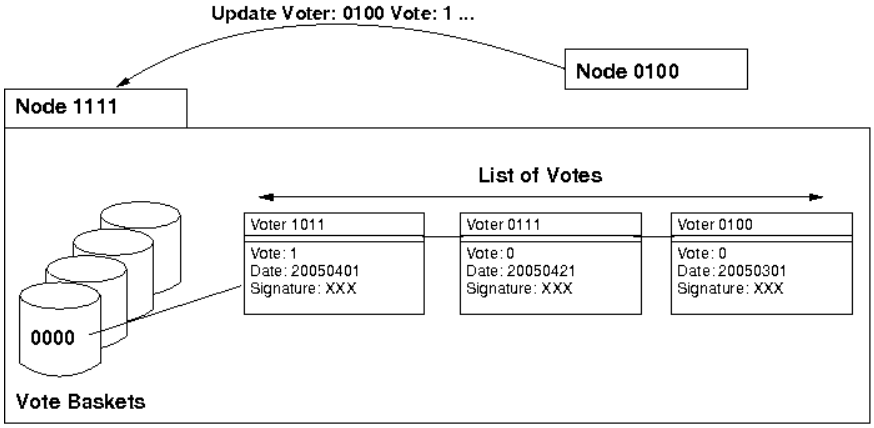


Figure 5.3: The Vote Basket needs to be updated

Due to the fact that votes fluetuate, the author suggests a short republish interval to keep the votes basket up to date. 60 minutes should be sufficient, the normal republish interval is 5 hours for sources and even 24 hours for everything else.

5.6 Retrieval of Votes

Retrieving votes from its associated storage peer involves three phases:

Retrieval of Votes After retrieving the ID of a potential download source j, the storage peer that holds the reputation information is computed, which is, in this example, $s_j = \neg j$. The storage peer is found via the `Find_Node` command. The Network then returns a list of Nodes "close" to the target ID. The fastest found node will be contacted with a `Find_Value` command. The information is retrieved.

Verification of Votes Phase two is vote verification. A possible solution that was done by P2PRep, is contacting each voter from the voting basket and validating the given vote directly. As this method created too much traffic on the network while delaying the verification phase, especially on low speed links this technique was replaced, by using digital signatures. While the notable gain of speed an simplicity may not be underestimated, once again the need for *assymetric cryptography* and a decentralized distributen of public keys is eminent. Every peer needs to know the public key of peers he wants to interact with. Otherwise a secure vote verification would not be possible. Fortunately, existing clients already support this for built in credits systems. This implementation specific part will be touched in the implementation specific part in chapter 6. The sole existence of this system is not only a good basis for further work but proves this concept as solid. With the existence of a PKI the retrieval and management of public keys can be transferred to the existing systems. Therefore, upon receiving a set of reputation information, the integrity of data can be verified by checking the signature that is contained in the votes. If it yields a positive result, the timestamp of the vote needs to be checked. If the second check succeeds, the vote is verified and can be further processed.

Decision The appliance of a policy to the collected and verified votes forms the final step before downloading. The outcome will decide on the trust relation between both peers, which ranges from *trust* to *distrust.* P2PRep presents in its paper different choices to do so. Indeed the selection and implementation is completely client specific and does not need a global defined algorithm. A fundamental basic approach was choose for this solution, that can be extended on behalves of special requirements.

The first step of the policy is to combine the collected votes with the credibility repository θ defined in chapter 5.2. This process will be called *weighting.* Using the aggregation operator every vote v_{ij} of voter i is multiplied with its credibility c_i. The result of the operation $v_{ij} \cdot \phi(\theta_i)$ defines the set of votes that will be used to compute the trust relation. As both votes and the operator ϕ are defined to $\{0, 1\}$ votes from untrusted sources will be dropped.

The policy can now decide on the the trust relation, using the weighted averages technique. For this purpose the votes are summed and divided by the number of votes n, producing g_j the global trust value.

$$g_j = \frac{\sum_{i=1}^{n} v_{ij}}{n} = \begin{cases} \geq 0.5\ trust \\ distrust \end{cases} \tag{5.3}$$

If these yield a positive result, j is contacted. Otherwise the search for another download source should be initialised. It may be argued that the system gets more secure if transactions are carried out only with peers that have a higher global trust value. We object against this, as peers with no reputation information have to be treated equal, as this would otherwise bully clients joining the network for the first time.

5.7 Extended Reputation System

In the previous chapter, the basic capabilities have been added to the voting system. To enhance the functionality even further, the different subjects that are involved in the process should be reconsidered:

- A trivial subject, when using a reputation system is the peer, that this transaction is about. We may or may not have had previously interacted with that peer before. After a transaction one can express a rating and update the local notion of that subject. Then the systems global view of that subject will be updated.
- When retrieving votes, there will be one or more ratings from other users. These should be weighted depending on previous performance. After a transaction, *voters* can easily be rated, because it is clear to which votes the client agreed and which he disagreed. Judging the quality of their votes by adding previous experiences adds a "history" feature to the algorithm.
- Also, the retrieval of votes is an interesting task. Malicious peers may try to hide their operations by giving reliable votes and even may carry successful transactions out. But in an attempt to pollute the voting system it is possible that they play unfair in their role as *storage peer*.

By looking at the different subjects involved in the process, another credential was identified: the *reputation as download source*. We could identify peers that try to pollute the voting system. While we have tried to assure a maximum of security for votes, it has already become clear in chapter 5.3 that a node dropping votes is a flaw that can currently not be handled. A voter could, however, periodically check the nodes, that previously received a `Store` command, whether they still return his vote. Furthermore, other clients could detect malicious storage peers using the cryptographic hashes. Also in this phase the clique detection and decision techniques from [DdVPS03] are extremely useful. They check the source of a vote for its IP address and try to resolve the IP range. If all votes come from the same subnet, a clique is highly probable.

6 Prototype Implementation

In this chapter, the integration work that was done to add reputation sharing capabilities to the open source eMule client, will be introduced. The first subchapter will present the existing architecture of the client, further subchapters will present the extensions to this architecture and the protocol used by the client. As malicious users may tend to circumvent protection, trying to attack clients more easily by pretending to be a high trusted user, the system requires an analysis of security considerations, which will conclude this chapter in subsection 6.4.

6.1 Architecture

The eMule client, which is at the time of writing at version 0.46a, started as an implementation of the EDonkey2000 protocol in 2002. It has received much attention, as it was the first open source client, that implemented it. While providing compatibility to the already existing client, it introduced a far number of improvements. These include a credits based queue system, which will be analyzed in paragraph PKI. It further overcame the limitation of the EDonkey network, which is based on central server, by providing a C++ implementation of the Kademlia protocol, based on Maymounkov's reference implementation in Java. The subsystem is cleanly separated from the eMule core and encapsulated in a own `Kademlia` namespace. The specification in [MM02] is closely followed, although the system is based on 128 bit identifiers, instead of the proposed 160 bit, as 2^7 is an integer on the basis of two.

The sourcecode of the client including the eDonkey2000 protocol already consists of 428 source and header files in the code's top level directory. The Kademlia implementation, found in the `kademlia/` subdir adds 48 files. The classes, that were added during this work can be found in the files `kademlia/kademlia/vote.{cpp,h}`, extensions to the source code are distributed across the Kademlia subdirectory. Grapical User Interface related changes were integrated into files located in the root directory of the source code. The nomenclature across the sourcecode is unique. Every class is implemented using a source and header file, that are named using the contained classes name. For example the class `CKadUDPListener` is implemented in the `UDPListener.{cpp,h}` files.

The task of integrating trust management into an existing client requires an understanding of its architecture. For this purpose, the components shown in figure 6.1 will be described in the following section.

UDPSocket All protocol communication is done via UDP. A transmitted message will be received by the `CClientUDPSocket::OnReceive()` callback of the windows API. The client implements both the EDonkey and Kademlia protocol. If the message starts with a magic number associated with Kademlia, like in figure 6.2, it will call the `CKademlia::ProcessPacket()` function, which forwards the request to the UDPListener instances `CKademliaUDPListener::processPacket()` function. As messages may be compressed, the client might previously decompress the encapsulated data.

The `ProcessResponse()` function calls appropriate functions on behalf of the `OPCODE` field of the protocol. Valid opcodes include `KADEMLIA_PUBLISH_REQ` or `KADEMLIA_SEARCH_REQ.` The first of both opcodes branches into the index service, while the second returns the results of a search. The component is the central point to send messages via the `sendPacket()` call.

Routing The routing service, offers all services described in the technical introduction in chapter 4.2. It adds Nodes to the internal routing bins and returns the IP address of a given contact. It is also the central place to receive the contact list from the system, via a call to `getAllEntries()`. All known contacts will be

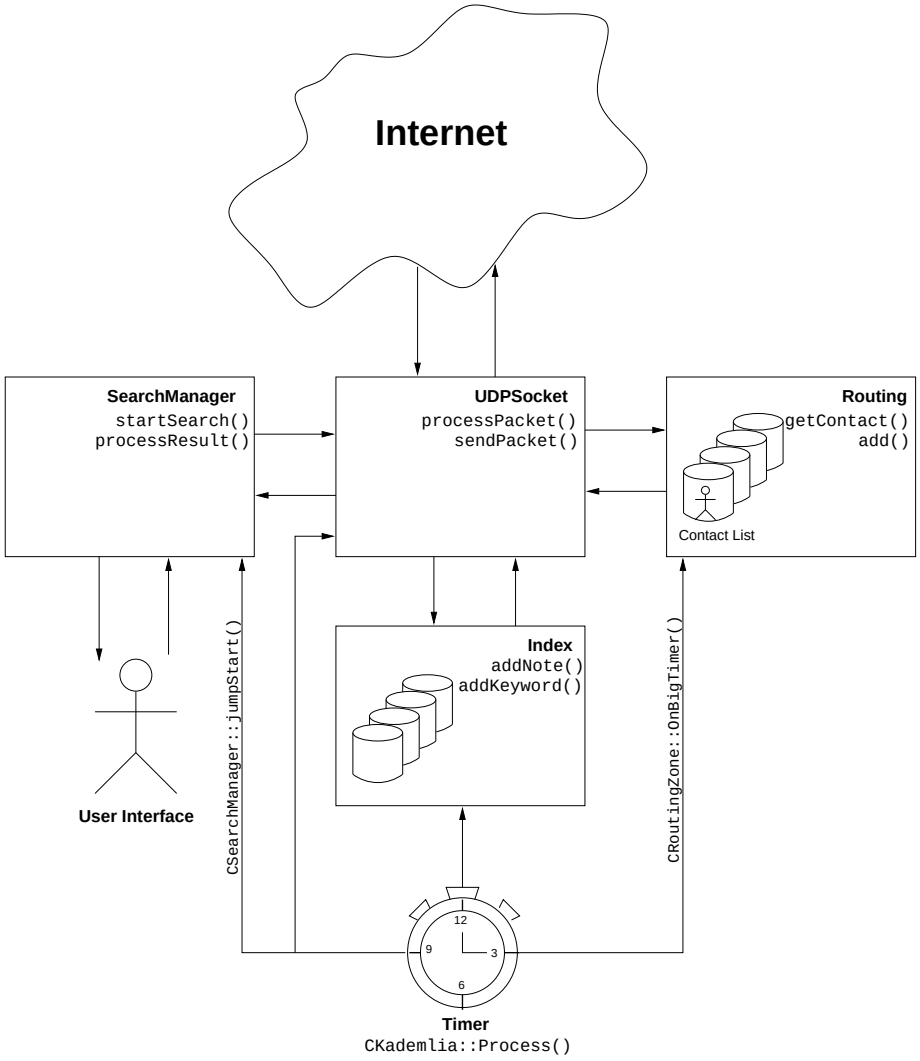


Figure 6.1: The components of the Kademlia implementation in the eMule client

```
+-------------------+--------+------+
| OP_KADEMLIAHEADER | OPCODE | DATA |
+-------------------+--------+------+
```

Figure 6.2: The header and opcode of a Kademlia packet

stored in this list. The graphical user interface of the client uses these functions to display a list of all known contacts in the upper part of figure 6.14. The complexity of the routing system is hidden behind a simple API, but unfortunately the search for closest nodes is not cleanly separated from the general query implementation. Routing messages that are sent across the network to discover the k nearest nodes to a key have the Opcode set to `KADEMLIA_REQ` replies are identified with `KADEMLIA_RES.` To get a better understanding of the routing elements in a query have a look at the explanations in the following "Search Manager" and see figure 6.3.

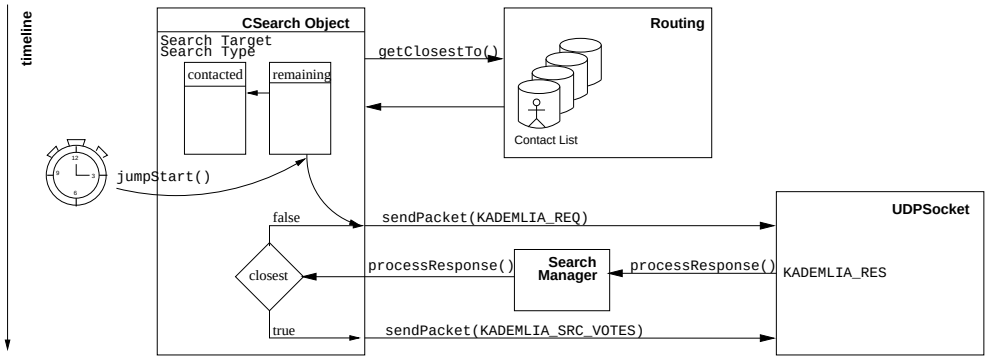


Figure 6.3: The routing part of a search request. A list of close nodes to a target is retrieved and subsequently contacted. If no closer nodes can be found the algorithm terminates.

Index The clients server capabilities are implemented in the Index subsystem. It provides the storage part of the distributed hash table. Data identified by its keys is addressed and provided in the internal data structures of this index. When the client encounters e.g. a `KADEMLIA_PUBLISH_REQ,` the index service is contacted to publish the given data at the given keywords location using a function `addKeyword().` Currently, clients can add themselves as *source* for a given key, register *keywords* or leave *notes* associated with the corresponding hash key. The data structures used for storing key/value pairs will receive attention in chapter 6.2.

Timer Besides events from the user interface and the UDP based network, a third instance provides events, that require actions from the system. The timer subsystem periodically checks for various conditions. It ensures, that both the provided sources, keywords and notes are kept consistent by republishing them on change or in the defined interval and deletes on its server capabilities key/value pairs that have expired and have not been republished in time. Various other timers keep the routing bins up to date using a function called `CRoutingZone::onBigTimer().` As described in the specification, it is done by sending `Ping` messages if entries are near expiration, or delete bins when nodes seem to be down after a given period of time. It also calls a method in the SearchManager called `jumpStart(),` that checks if search results have arrived and provides them to the associated `CSearch` object.

Search Manager Both searches that are initiated by the user - e.g. searching for a keyword - and those by the system, e.g. node lockups, are registered at a central instance: the "Search Manager". This object, identifies all searches, using the associated hash key they are searching for. This technique enables it to

receive packets from the Kademlia UDP listener and forward it to the appropriate `CSearch` object, while explicitly preventing simultaneous searches for the same key.

The interface for creating a search inside the SearchManager is the function `prepareLookup()`. The function requires a "search type", a bool value, that controls if the search should also be started immediately and the search key, as arguments. A new search object is created and if advised to do so, the `CSearch::go()` method is called, which starts the routing part of this query. k nodes closest to the passed key, will be located by calling the function `CSearch::sendFindValue()`. The routing messages, marked with `KADEMLIA_REQ` will be sent and `KADEMLIA_RES` messages will be received. These routing messages will be passed to the `SearchManager::ProcessPacket()` function, which forwards them to `CSearch::ProcessPacket()` that will contact the returned closer nodes. If no closer nodes can be found, the routing algorithm terminates and the actual query will be sent to the discovered nodes. These packets will have an opcode set to the appropriate type of the query, as described in the previous UDPListener paragraph. Results for this query will finally be forwarded to the search via the `processResult()` function. There is also a graphical representation, as seen in the lower part of figure 6.14.

PKI Among the components that were shown in figure 6.2, the used client has already asymetric key cryptography and functions to retrieve the public key from another peer, which added additional value to the platform. In previous chapters, it was analyzed, that *identity trust* is the fundamental basis for *Provision Trust*. Furthermore subchapter 5.3 explained the use of cryptographic techniques in this solution. This implementation relies on self signed keys, without the use of a Certificate Authority. Clients exchange keys the first time they make contact. The current use of these keys is a credit system that rewards other users: eMule uses a key handshake method to ensure the authenticity of this user. Users assign credits to each other, depending on the volume of the transferred data between them. Credits stored for a user are only granted if this authentication has been successful.

> *The strict queue system in eMule is based on the waiting time a user has spent in the queue. The credit system provides a major modifier to this waiting time by taking the upload and download between the two clients into consideration. The more a user uploads to a client the faster he advances in this client's queue. [Pro06]*

The existence of a repository for keys of other users and standardized ways of exchanging keys, allows this implementation to be based on existing services. The client uses the Crypto++ library from *http://www.eskimo.com/~weidai/cryptlib.html*. The library is a C++ interface to various one way hashes, including `SHA-1, MD2, MD4, MD5` and both symmetric and asymmetric cryptography functions. The library makes extensive use of templates. The client's cryptographic functionality is implemented in the file `ClientCredits.h`. The algorithm used in the client is the RSASSA-PKCS1-v1_5 algorithm described in RFC2437 [KS98]. The signature standard is therefore PKCS #1, SHA-1 is the chosen hash function. On startup the client reads the keys from the file `cryptkey.dat` and stores the private key in a datastructure called `m_pSignkey` of type `RSASSA_PKCS1v15_SHA_Signer`. The public key is of type `RSASSA_PKCS1v15_SHA_Verifier` and is written in a byte array called `m_abyMyPublicKey`. If the keys are not available, a new keypair is created on startup.

For the intended usage of this work, signing and verifying votes, two functions were added to the `ClientCredits` class:

Listing 6.1: The function definitions for signing and verifying Votes

```
uint8 SignBlock(uchar* pachOutput, uint8 nMaxSize,
                Kademlia::Vote &t);
bool  VerifyBlock(CClientCredits* pTarget,
                  const uchar* pachSignature, uint8 nInputSize,
                  Kademlia::Vote &t);
```

These functions both take as an argument a `Vote`. The function `SignBlock` will write a signature of the passed vote to the temporary buffer `pachOutput`, not exceeding `nMaxSize`. It sets up a random number generator from the Crypto++ library, serializes the vote into a `byte[]` array and calls the `PK_Signer::SignMessage` function. The `PK_Signer` interface is implemented inside the

RSASSA_PKCS1v15_SHA_Signer class. The signature will then be stored in the callersupplied buffer. As it will become clear in the course of this subchapter, the caller will proceed by creating a special kind of CTag storing the signature and append it to the Vote.

The verification of a received Vote is done in similar fashion. Before calling VerifyBlock, the client needs to remove the signature from the vote. This is convenient, as the signature was not contained in the vote block at the time of signing the vote. Both are passed separately to the function. The public key of the voter is retrieved from his CClientCredits structure. With this information, a new instance of the class RSASSA_PKCS1v15_SHA_Verifier is created. The Vote is written into a temporary buffer and the function VerifyMessage is called, which will yield a positive or negative result.

6.2 Integration into the eMule Client

The approaches and solutions used will be presented here. The previous chapter introduced the key components, that need to be adapted, when adding the proposed solution. Serving votes requires additions to the protocol that enable sharing and retrieving of votes and the index to provide the serving capabilities of votes and the contacts list, to store the users own experiences with the contact. Furthermore, timers are needed, that republish the vote information. Finally, a graphical user interface needs to be added. Figure 6.4 reflects the required changes to the infrastructure.

UDPListener The implementation required four new opcodes. The first two denote a request to publish votes KADEMLIA_PUB_VOTES_REQand the appropriate reply opcode KADEMLIA_PUB_VOTES_RES. The remaining opcodes are used for searching and retrieving a set of votes from a given key are called KADEMLIA_SRC_VOTES_REQ and KADEMLIA_SRC_VOTES_RES. The processPacket function of the class CKademliaUDPListener was extended to branch to four additional created functions:

```
KADEMLIA_PUB_VOTES_REQ --> processPublishVotesRequest()
KADEMLIA_PUB_VOTES_RES --> processPublishVotesResponse()
KADEMLIA_SRC_VOTES_REQ --> processSearchVotesRequest()
KADEMLIA_SRC_VOTES_RES --> processSearchVotesResponse()
```

Furthermore a helper function was defined, implementing the mapping function from chapter 5.4, that returnes the binary negative of a given KAD CUInt128 identifier. It is call called contactToNode.

Listing 6.2: The definition of the mapping function

```
CUInt128 contactToNode(CUInt128 &contact);
```

Experience Repositories and Vote Structure For the implementation of this work, two incarnations of a vote exist. The first is the direct representation of the repositories θ and Ψ . The second representation is the global trust value $\phi(\Psi)$, along with additional information.

To represent both the reputation and the credibility repository a new type of object was designed in the Kademlia namespace, the ExperienceRepos class. This class is designed to provide all functionality the client requires to use and contribute to a reputation system. A part of the public interface of the class is defined as the following:

Listing 6.3: The class ExperienceRepos

```
class ExperienceRepos
{
    friend class CVotePage;

public:
    ExperienceRepos(CUInt128 key);
```

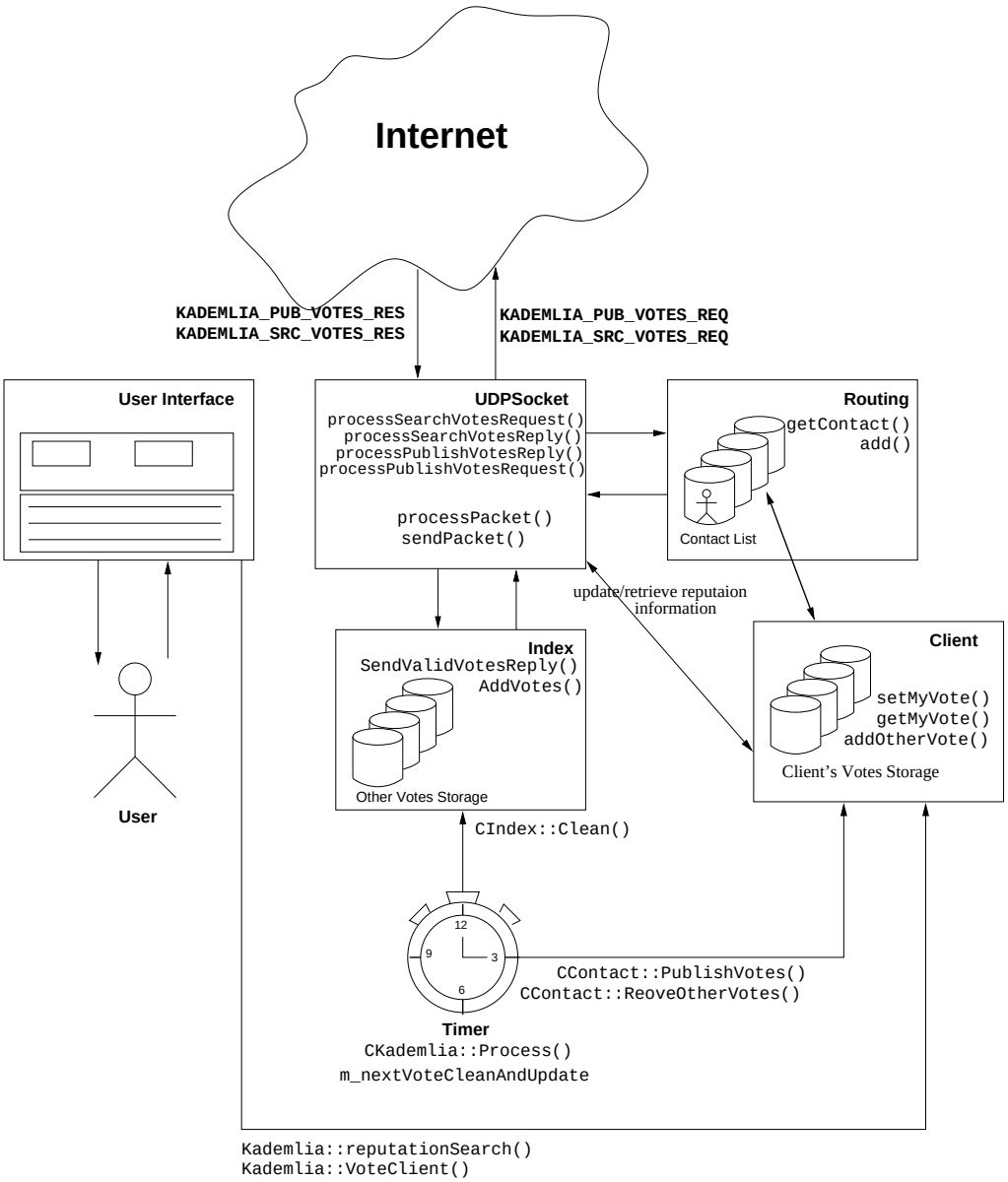


Figure 6.4: An extended architecture, allowing reputation sharing

```
    ~ExperienceRepos();

    uint8 getReputation();
    uint8 getCredibility();
    uint8 getVoteSourceRep();

    void alterReputation(uint8 how);
    void alterCredibility(uint8 how);
    void alterVoteSourceRep(uint8 how);

    bool isValid();
    bool isModified();
    Vote &getVote();

  protected:
    uint8 getReputationRep(uint8 &pos, uint8 &neg);
    uint8 getCredibilityRep(uint8 &pos, uint8 &neg);
    uint8 getVoteSourceRep(uint8 &pos, uint8 &neg);

    uint8 setReputationRep(uint8 pos, uint8 neg);
    uint8 setCredibilityRep(uint8 pos, uint8 neg);
    uint8 setVoteSourceRep(uint8 pos, uint8 neg);

  private:
    uint8 getGlobalTrustValue(uint8 pos, uint8 neg);

};
```

The class is implemented as described in the previous chapters. Repositories of positive and negative performance can be accessed by its public interface. The calls to `getReputationRep()` and similar functions return the positive and negative baskets, while the `setReputationRep()` class of functions overwrites them. These functions can only be called from a specially authorized `friend` class. Currently, only one friend class, that is tied to the graphical user interface, described in chapter 6.3, may access these functions, the `CVotePage` class.

To retrieve the previously described *global trust value*, a set of functions are provided, starting with CKademliaUDPListener `getRepuation()` that return a eigth bit integer. These functions currently call all the private `getGlobalTrustValue()` function, that computes its result on behalf of the policy defined in chapter 5.2. As it has been decided in the current implementation, only zero, for a negative recomondation, or one, for a positive recommondation, is returned, which could easily be extended.

By providing this critical functionality encapsulated in a single class, a client that wants to alter the behavior of the system, just needs to subclass and overwrite specific methods. To alter the behavior of every trust decision, the function that returns the global trust value could be overridden, for finer grained access, the repository specific functions, like `getCredibility()` could be altered.

The function `getVote()` needs some explanation. The signature returns a reference to a data structure called `Vote` which is also called a `TagList`. It stores the global trust value, along with the required meta data, as described in subchapter 5.3.

The class `TagList` is a list from the standard template library, whose elements are of type `CTag`. This abstract class is used in the client to store data of a defined type and identified by a name. Commonly, every kind of published data in the **Index** subsystem is available in this form. For easier management of these lists, a class `CEntry` is available, which provides searching facilities, along with records of the IP address, the source port and the ID of the client that transmitted the data. TagLists can easily be serialized[1] to a binary representation, that can be stored in a file, or transmitted over the network, using simple APIs provided by the client. The available datatypes can be extended by subclassing the `CTag` class. Types include

[1] For the serialized representation of a TagList, proceed to subchapter 6.2.1

strings, integers, or bytes. The widespread use and the simplicity made us choose to implement the second representation of votes in this form.

For an example on how to use tags, the following will define a tag needed for reputation sharing and storing the global trust value $\phi(\Psi)$. In subchapter 5.2 it was agreed upon, that mapping it to an eigth bit integer is more than sufficient, for it currently has only two states. The `CTagUInt8` subclass is therefore chosen. Furthermore a name is needed to identify the stored data. `TAG_REPUTATION` was chosen as identifier. To access a received global trust value, a list can be scanned for this identifier. A figure might help to explain the relation between tags and entries.

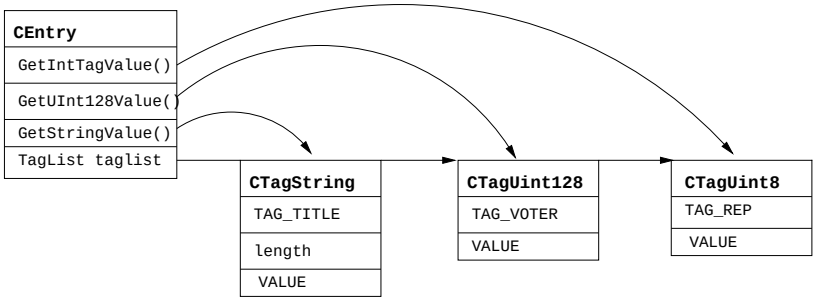


Figure 6.5: A CEntry is a datastructure, facilitating the access to a TagList

Besides a tag for reputation values a set of additional tags was defined, to comply with the requirements for votes from subchapter 5.3. The complete set of tags is:

```
TAG_REPUTATION
TAG_CREDIBILITY
TAG_VOTESOURCE
TAG_TIME
TAG_SIGNEDTUPLE
TAG_VOTER
TAG_VOTED
```

The values of the tags are eigth bit integers for the measurements of performance, 32 bit for timestamps and 128 bit integers for the IDs. The SIGNEDTUPLE represents the digital signature, which is a `byte[]` value.

Contact List After becoming clear on the representation of a vote, an association between the `CContact` class and the trust data was made by adding methods and attributes to the class.

Listing 6.4: Functions, added to the `CContact` class

```
bool PublishVotes();
bool RemoveOtherVotes();

void ScheduleOtherVotesRemoval();

ExpiriencesRepos *getMyVote();
void setMyVote(ExperienceRepos *x);

void  addOtherVote(Kademlia::CEntry* pEntry);
const VoteList& getOtherVotes();
uint8 getOthersTrust()

VoteList m_OtherVotes;
time_t m_lastPublishTimeKadVotes;
time_t m_removeOtherVotesBasket;
```

Both of the first two functions, are needed for the timer interface described in the next paragraph. The `{get,set}MyVote()` functions return the associated trust repositories that were described in the previous

chapter. If a reputation search was initiated via the Kademlia wide global function `reputationSearch()` the processing of incoming results will add received votes via the `addOtherVote()` function. Once again, an object of type `CEntry,` which facilitates the processing of the contained `TagList, is passed.` To retrieve all currently known votes a list of `CEntrys` can be retrieved by calling `getOtherVotes().` The `time_t` variables are used to monitor the expiry of votes. After publishing a vote, the timer `m_lastPublishTimeKadVotes` is set appropriately and is accessed in the `PublishVotes()` function, to determine, if the published votes need to be refreshed. In that case the function will return `true,` as illustrated in figure 6.6

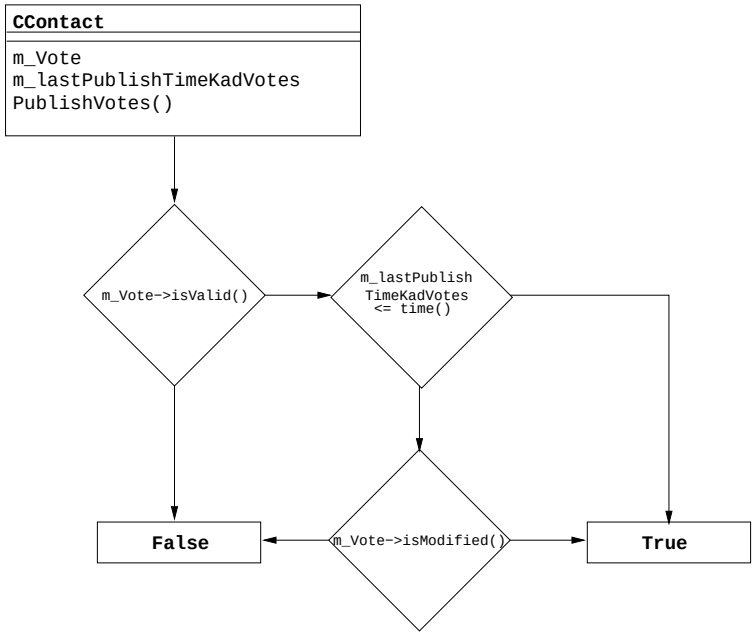


Figure 6.6: The callgraph of `PublishVotes()`

The second timer monitors the age of received votes. These will automatically expired, to ensure that only valid votes will be used as base of a trust decision, via the function `ScheduleOtherVotesRemoval()`. The timeframe of this decision was set to be 15 minutes.

```
#define KADEMLIARVOTEXPIRY              MIN2S(15)
```

After this interval the votes need to be received again from the storage peer. This timeframe was chosen, to allow a “cached” trust decision in the case of subsequent transfers with a peer, while still ensuring an up to date trust decision.

To compute the trust decision, another function is provided which analyzes and weights all votes according to the global credibility values each voter is assigned. As described in 5.6, the function `getOthersTrust()` comes up with a trust recommendation, based on the public available information.

Timer An additional timer is needed for the implementation that expires, votes from the Index and prevents client’s own votes from expiring, from the global index, by republishing them. As previously described the main timer routine is `CKademlia::Process().` For this purpose another timestamp was introduced, with the signature `time_t CKademlia::m_nextVoteCleanAndUpdate.` that is checked in intervals of `KADEMLIAREPUBLISHTIMEV.`

```
#define KADEMLIAREPUBLISHTIMEV  HR2S(1) //1 hour
```

That timer loops through the list of contacts, by calling appropriate functions of the routing subsystem. Each contact is asked to `PublishVotes(),` which was already described. It further cleans up the received votes of other uses, that have been cached by a previous reputation search. Then the next contact is checked.

The index subsystem is contacted via the `CIndexed::clean()` function, which expires votes, that were explicitly published at this node.

The timer runs in an interval defined by the `REC_VOTES_DELETE` variable, which is set at compile time of 20 seconds.

Index In figure 5.3, this part of the index system was already sketched. A set of so called baskets, one for every voted subject, stored on the node, should hold a list of voters for that subject. Instead of a list, a set of hash tables are used in the system called `SrcHash`, that use voters ID as key, providing a pointer to a `CEntry` representing the data. To be able to quickly retrieve the voters hash table associated with a vote subject all hash lists are added to a `std::map` derived data structure.

Listing 6.5: The data structure storing votes

```
CMap<CCKey,const CCKey&,SrcHash*,SrcHash*> m_VotesMap;
```

This map stores the ¬ IDs of the subjects of the votes as keys and returns references to the previously described hash tables containing their votes. Figure 6.7 reflects these changes.

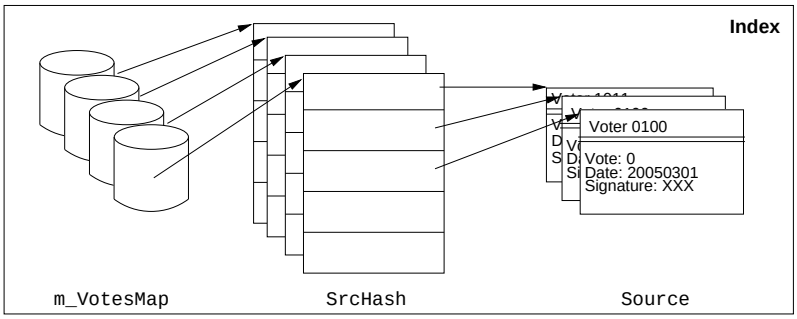


Figure 6.7: Data structures involved in the implementation of Vote Baskets

To add votes to the index, the data set is manipulated via the `AddVotes()` method. This method is called from the UDPListener in response to a `KADEMLIA_PUB_VOTES_REQ.` Before adding the received data to the list, the function will check the distance between the node's own ID and the key, to prevent storing values at wrong locations in the distributed hash table. It will also require a creation time encapsulated in the vote, which is used to check if the vote is still valid. This prevents replay attacks, as they will be described in subchapter 6.4.

When the data has been validated, the `m_Votes_map` is queried for the given key. In the trivial case, the query yields no result and a new `SrcHash` is added to the map, containing only one `Source` entry, identified by the given voters key. If the map returns an existing hash table, but the voter is not already contained, a new `Source` data structure is allocated. The object references the `CEntry` containing the actual vote and then added to the hash using the ID of the voter as key. If both already exist, the vote is considered to be more recent than the previous vote, which is ensured by checking both `TAG_TIME` tags. If the vote is more recent the previous version is discarded and replaced by the newer data. If for some reason a older vote was received, almost certainly some kind of fraud was detected.

To return a set of votes to a given key, the function `SendValidVoteResult` is provided. If a lookup in the map yields a hash table, a packet containing all available votes is constructed and sent to the query issuer. The detailed assembly of these packets will receive attention in the protocol description in the next subchapter.

As a termination of the program would otherwise remove the indexed votes, a file `vote_index.dat` is written on program termination in the destructor `CIndex::`$\sim$`CIndex`, storing votes permanently on disk. On startup the `readFile()` method reads them back into memory, by adding them via the `AddVotes()` method, that ensures the validity of the vote, by checking the `TAG_TIME` of the vote. So no outdated votes will be distributed to clients.

Searches The search API of the client is generic, so that no changes are required to the core system itself. The remaining task in this subsystem was to add two new search types that allow a specific processing of results, on search termination. The types `VOTES` and `STOREVOTES` were added in the namespace `CKademlia::CSearch.` When receiving a query, the type of the search is checked for these additions. If a search with type `STOREVOTES` terminates, no further action is required, as this means that a vote has successfully been published. If results for a votes query are received, they need to be checked for validity and added to the appropriate `CContact` structure, by calling the `addOtherVotes()` function of the object. This function will search for the voter in the internal `m_otherVotes` data structure. If an entry is found, it is assumed, to be outdated and is deleted, after the proposed vote has been verified. If a dialog, that displays the key's reputation information, is currently open, it is informed of the addition of data.

6.2.1 Protocol Description

There have been four additions to the already implemented protocol. Their binary representation and the further processing will be introduces in the following.

CTags CTags were chosen to represent votes inside the clients. For transmission purposes these need to be serialized. Therefore, we will start with the binary representation of CTags in figure 6.8, to be able to describe them in the rest of the subchapter using their name attribute. A TagList will be serialized in the same way, but the packet will be preceded by the number of tags, that will be sent.

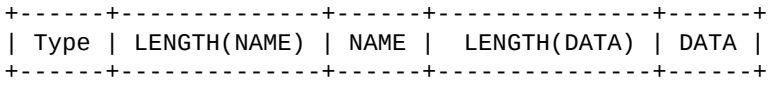


Figure 6.8: Serialisation of a CTag

Retrieval of Votes Normally a search for votes is initiated on behalf of the user. This high level functionality is provided by the `repuationSearch` function, defined in the `vote.h` header file. This function creates a `CSearch` object and sets its type to VOTES. Using the `contactToNode()` function it computes and sets the search key. If no other query for the key is currently active, the call to `SearchManager::startSearch()` succeeds, otherwise an alert informs the user about the running search, asking him to repeat the query later. The search object will first try to locate the storage peers nearest to the key. As previously described, `KADEMLIA_REQ` packets are sent for this purpose. When this routing phase succeedes, the closest nodes are contacted with `KADEMLIA_SRC_VOTES_REQ` packets, containing the search key and the queries own ID. The packet will look like figure 6.9.

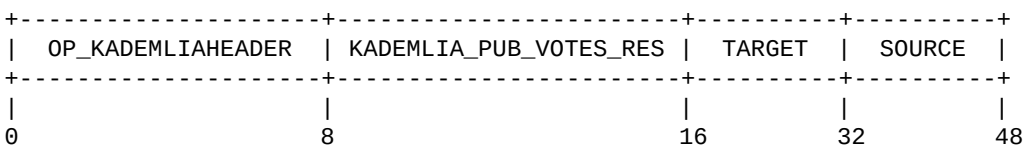


Figure 6.9: The elements of a search votes request

On the server side a storage peer will receive the packet in the UDPListener and branch into the function `processSearchVotesRequest().` It will extract the queried key and the source of the query. Then it will contact the index subsystem by calling `SendValidVoteResult()` along with the source and the search key as parameters. As described in the previous paragraph, the the index will look up the key in the datastructure `m_Votes_map.` If no results are found the algorithm terminates and no response packet

is produced. Otherwise it iterates over all elements in the hash map and produce packets with 50 votes at maximum. Figure 6.10 illustrates these packets.

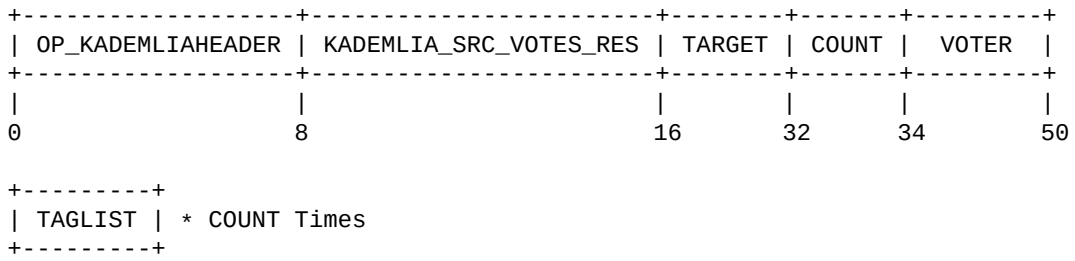


Figure 6.10: Elements of a search votes reply

When the client, that inited the query receives this packet it will pass it to the `SearchManager::processResult()` function. Using the search key, the search object that sent the query is located and the packet is further passed via the `CSearch::processResult()` function. Using the objects search type, that was set when creating the `CSearch` object the function calls `processResultVotes()` to complete the processing.

This function, calls addOtherVotes() to add each received vote to the `CContact` structure. It further notifies an open Votes dialog of the new arrived data, by calling `CContact::UpdatedData()`. This triggers a `UM_DATA_CHANGED` notification, to be sent if a dialog displaying the reputation of this client is open. This will be discussed in detail in subchapter 6.3.

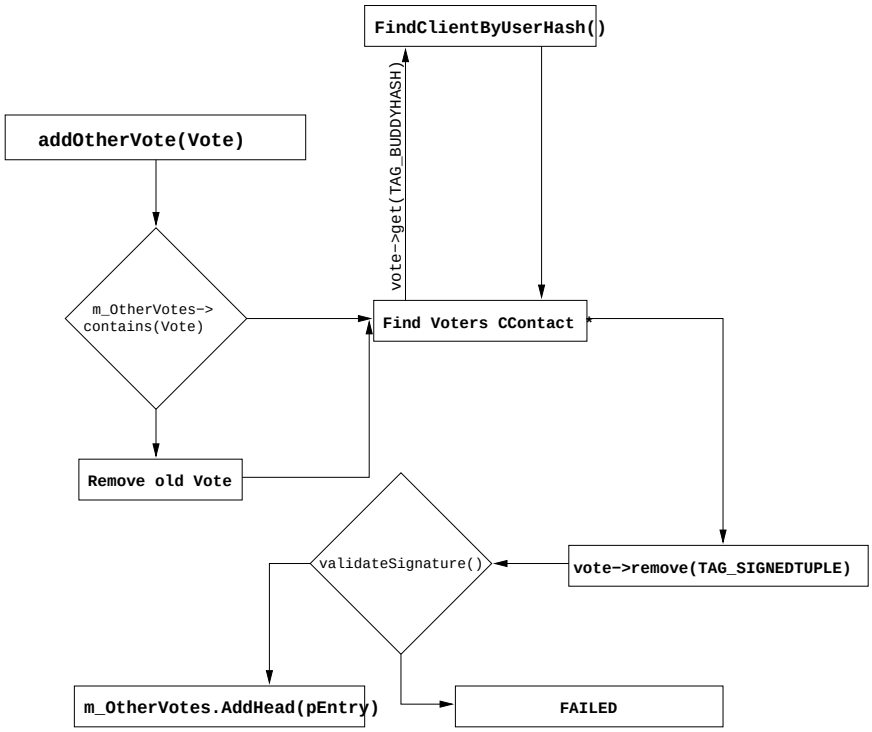


Figure 6.11: A callgraph of the addOtherVotes() function

Publishing of Votes The process of publishing a vote is initiated by the timer subsystem, described in the previous subchapter. If a republish interval is hit, or votes changed previously, the `PublishVotes()` function will return `true`. The `Kademlia::CSearchManager::prepareLookup` function, will

be called, to instantiate a `CSearch` object, setting the search type to `STOREVOTES`. The search subsystem will go through all steps described previously, calling the routing subsystem, in order to retrieve the closest known nodes, contacting them for closer nodes. If no closer nodes could be found, the `Vote::getVote()` function will be invoked, returning the signed vote. This will be sent to all *k* clients, that were found to be the closest nodes. The packet will include all fields, found in figure 6.12.

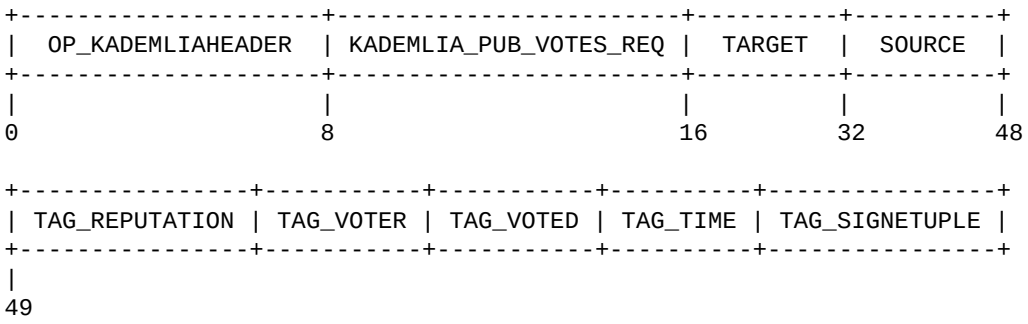


Figure 6.12: The elements of a publish votes request

A limitation of the eMule client forced us to include another element in the protocol, not included in the above figure. Since the client supports both the Kademlia and the eDonkey2000 protocol, the implementation requires a given eDonkey user ID to identify the stored public key of a client. After discussing the details of searching the complete list of public keys, it was decided, that the eDonkey ID, called *buddy hash* will be included in the vote for practical reasons. This hash is contained in the `TAG_BUDDYHASH` tag of the vote and can safely be removed from the system after the programming interface has been changed accordingly.

The storage peers will reply to the received packet with a packet containing an opcode set to `KADEMLIA_PUB_VOTES_RES`. This packet has attached load parameter that expresses how much data is already published at this node. Nodes will only publish a fixed amount of data. To make the network aware of this possible overload, a load parameter is returned after every publish request.

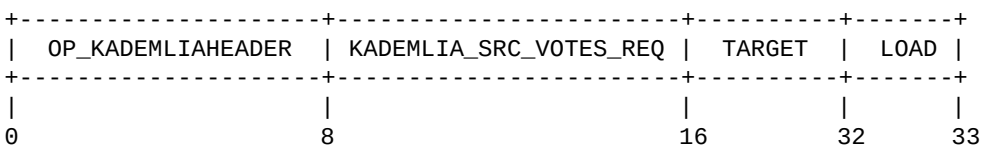


Figure 6.13: The reply to a publish votes request

6.3 User Interface

The actual user interface, that can be used to express trust in the credentials of another clients is implemented as a “Tab” page in a Dialog. This implementation was chosen over a single dialog, as this addition should remain extensible via different “snap ins”. It was Incorporated into the Kademlia section of the eMule client, using a new created context menu, that is illustrated in figure 6.14 on page 59.

The dialog itself shows up when a contact is selected in the Contact list in the Kademlia window and the “View reputation of this user” entry of the context menu is selected. Figure figure 6.15 shows the resulting window. It is divided into two sections. The first is entitled “Your vote”. This contains the experience repositories of the user, with this contact. The lower half of the dialog is called “Other User’s Votes” and presents a list of votes.

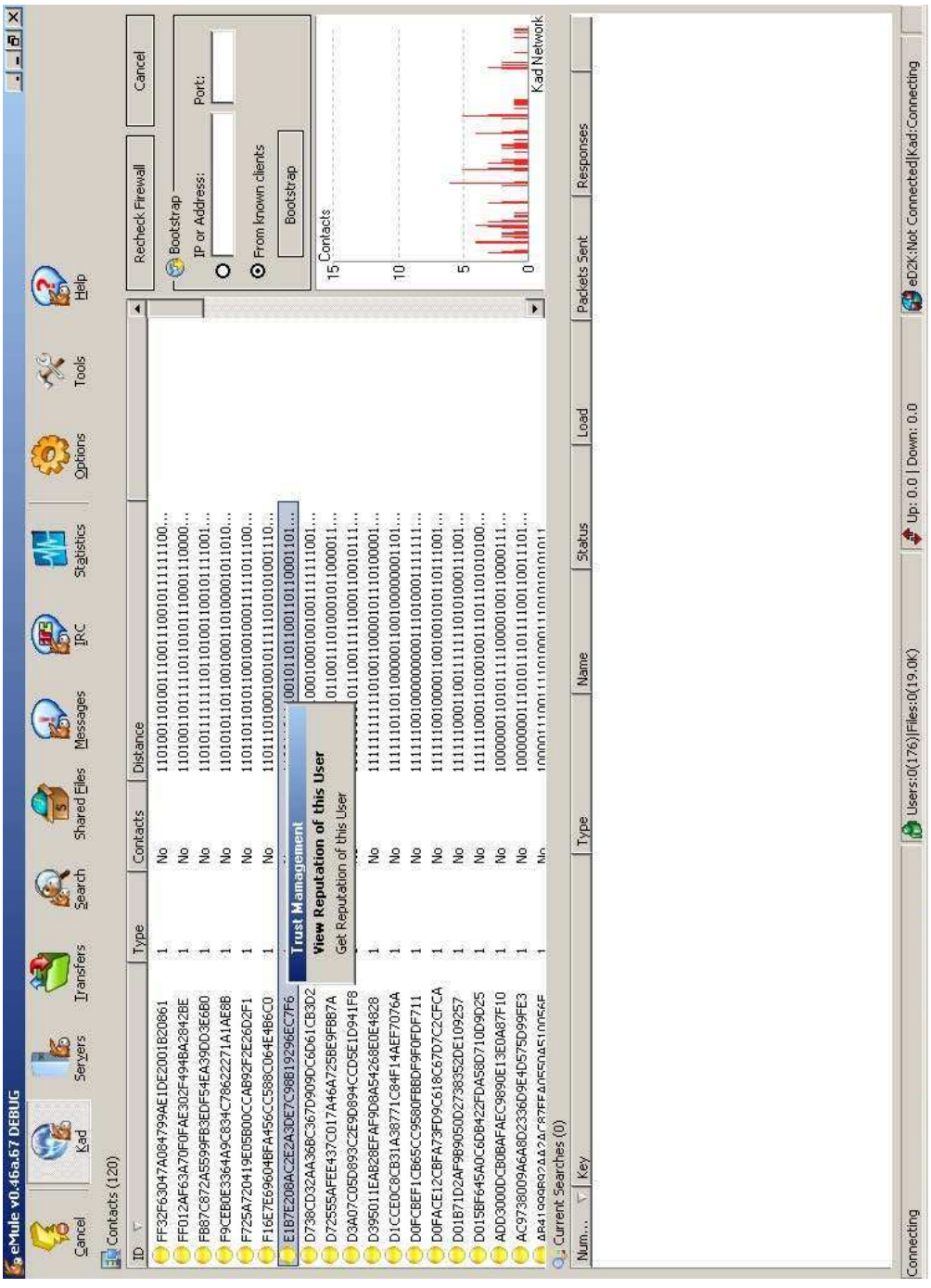


Figure 6.14: The Kademlia Window of eMule

In the experience repository section, users can express their trust in other users, by setting a positive negative ratio. Two editboxes, with associated spinboxes allow to manipulate the positive experiences, which are altered in the left box and the negative side, which is the right box. Next to the boxes, every line contains a trust recommendation. In the reputation line, the text can read “This client is trusted” and “This client is not trusted”, dynamically updated on the decision of the `getReputation()` function. If one of these values is altered and these changes are accepted by pressing the “Apply” or “OK” button, the vote is set to a “modified” state and will be republished on the next iteration of the timer subsystem, which checks every vote for being modified.

The list in the lower section of the dialog is set up using the `TAG_REPUTATION`, `TAG_VOTER` and `TAG_VOTESOURCE` data that is provided with votes. The list only shows currently cached votes. However, using the “Search KAD” button, a new reputation search is initiated. If results arrive while the dialog is open, the list is dynamically updated with the received votes. Below of the list of votes, another text label sums up the recommendations given by other users. It can be set to “Other Users recommend to trust this user” or the expression of distrust. On an update of the list, this text label will be computed again.

Implementation The user interface was designed using the guidelines and examples given by the eMule source code. The client uses the *Microsoft Foundation Classes (MFC)*, so all graphical user interface work was done accordingly.

The first task was to decide where to integrate a trust management user interface. As the Kademlia system, has an own window in the eMule client, that currently displays a list of all known contacts and active queries, it was chosen to add context menu to the contact entries.

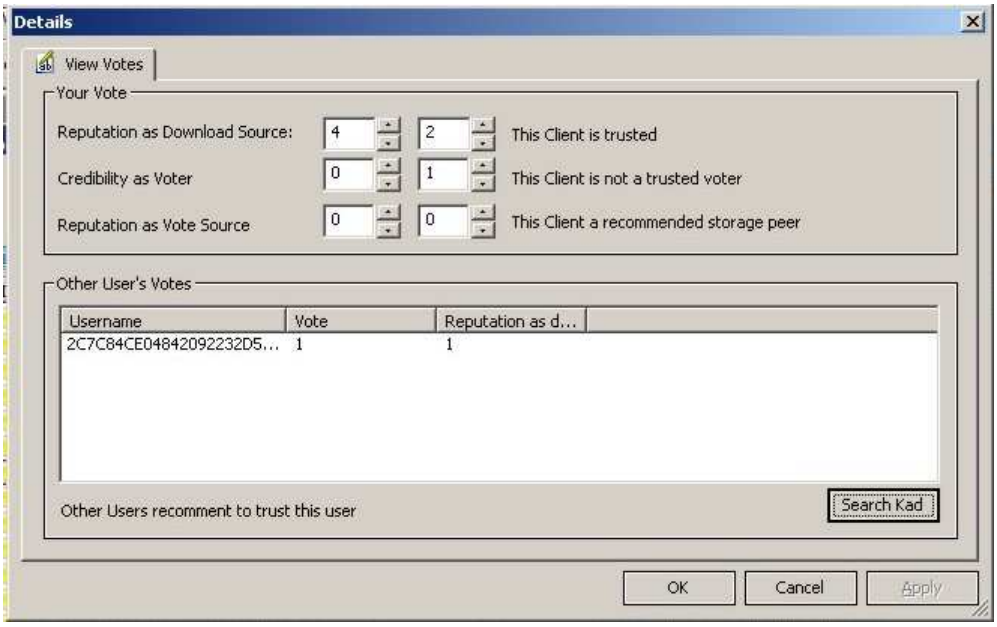


Figure 6.15: The new created vote dialog

The implementation of the dialog was done in similar fashion. Every control in the MFC is associated with a name, that can be chosen in the graphical user interface designed of *Visual Studio*. This dialog was named `IDD_VOTEDIAG`, which has to be passed to the constructor of the base class `CResizablePage`. Using this identifier the Windows runtime can load the dialog from the description in the resource file `emule.rc` and display it.

Every element in the User Interface is also associated with an arbitrary identifier. In the Microsoft Foundation Classes there are two possible ways to access these. The first is to directly access the element by calling a function inherited from the parent class, called `GetDlgItem()`. Passing the name of the desired control,

the runtime returns the associated object. The object representing the list, that shows the votes of clients is e.g. created with the following call:

Listing 6.6: Microsoft Foundation Classes supply functions to retieve pointers to GUI elements

```
CListCtrl *pList = (CListCtrl*)GetDlgItem(IDC_LIST);
```

This method is sensible for getting pointers to GUI objects. The second way is suited to keep local variables consistent with the displayed data is called *Direct Data Exchange (DDX)*. This technique is provided by the windows runtime. To use the DDX feature, a method must implement the `DoDataExchange` method, which specifies a mapping of GUI item names to local variable names. Macros for mapping controls to objects `DDX_CONTROL` and `DDX_TEXT,` mapping text to variables were used in the following example:

Listing 6.7: An example for Direct Data Exchange

```
  //Map first negative spinbox to control
  DDX_Control(pDX, IDC_SPIN_N1, m_repSpinNeg);

  //Map contence of edit box to variable
  DDX_Text(pDX, IDC_REP_EDT, m_RepVotPos);
```

DDX is automatically invoked by the windows runtime, however, if needed, the function `UpdateData()` is provided, for manual invocation.

For the user interface to work, the last action taken is the declaration of a *Message Map*. A message map is a central component of the Windows Operating System. Interprocess Communication (IPC) is done using messages. Drivers of *Human Interface Devices* generate events, that are sent to the central windows message queue. This queue is periodically checked for new data. If a new message arrives it locates the current active window and inserts the message in the message queue of the GUI thread. There are predefined actions depending on the type of the message. When providing a user interface, it is sensible to overwrite some of the standard handlers. This is done by overwriting parts of the standard *Message Map*.

The user interface presented in this subchapter is not very complex, but needs to react on some messages to ensure the validity of the entered data, or to react on changes in the trust values. The following is an export of the used map.

Listing 6.8: The Message Map of the vote dialog

```
BEGIN_MESSAGE_MAP(CVotePage, CResizablePage)
ON_WM_DESTROY()
    ON_MESSAGE(UM_DATA_CHANGED, OnDataChanged)
    ON_EN_CHANGE(IDC_DLVOT_EDT, OnEnChangeEdt)
    ON_EN_CHANGE(IDC_DLVOT_EDT_N, OnEnChangeEdt)
    ON_EN_CHANGE(IDC_CRED_EDT, OnEnChangeEdt)
    ON_EN_CHANGE(IDC_CRED_EDT_N, OnEnChangeEdt)
    ON_EN_CHANGE(IDC_REP_EDT, OnEnChangeEdt)
    ON_EN_CHANGE(IDC_REP_EDT_N, OnEnChangeEdt)
    ON_BN_CLICKED(IDC_KADBUTTON, OnBnClickedKadbutton)
    ON_NOTIFY(UDN_DELTAPOS, IDC_SPIN_P1, OnDeltaposSpinP1)
    ...
END_MESSAGE_MAP()
```

The first message `UM_DATA_CHANGED` is a IPC message, similar to a signal from the UNIX environments. The message was defined in the file `UserMsgs.h` and is used for notifying windows about an event. The message is sent from search objects, if votes on the currently displayed client were found. In response to this message the function `OnDataChanged()` calls `UpdateList(),` redisplaying the list of votes and the recommendation information on the bottom of the dialog.

`ON_EN_CHANGE()` is a notification of changed controls. The listed controls are the input boxes that represent the experience repository. Their associated function manually calls the DDX functions, to update the local

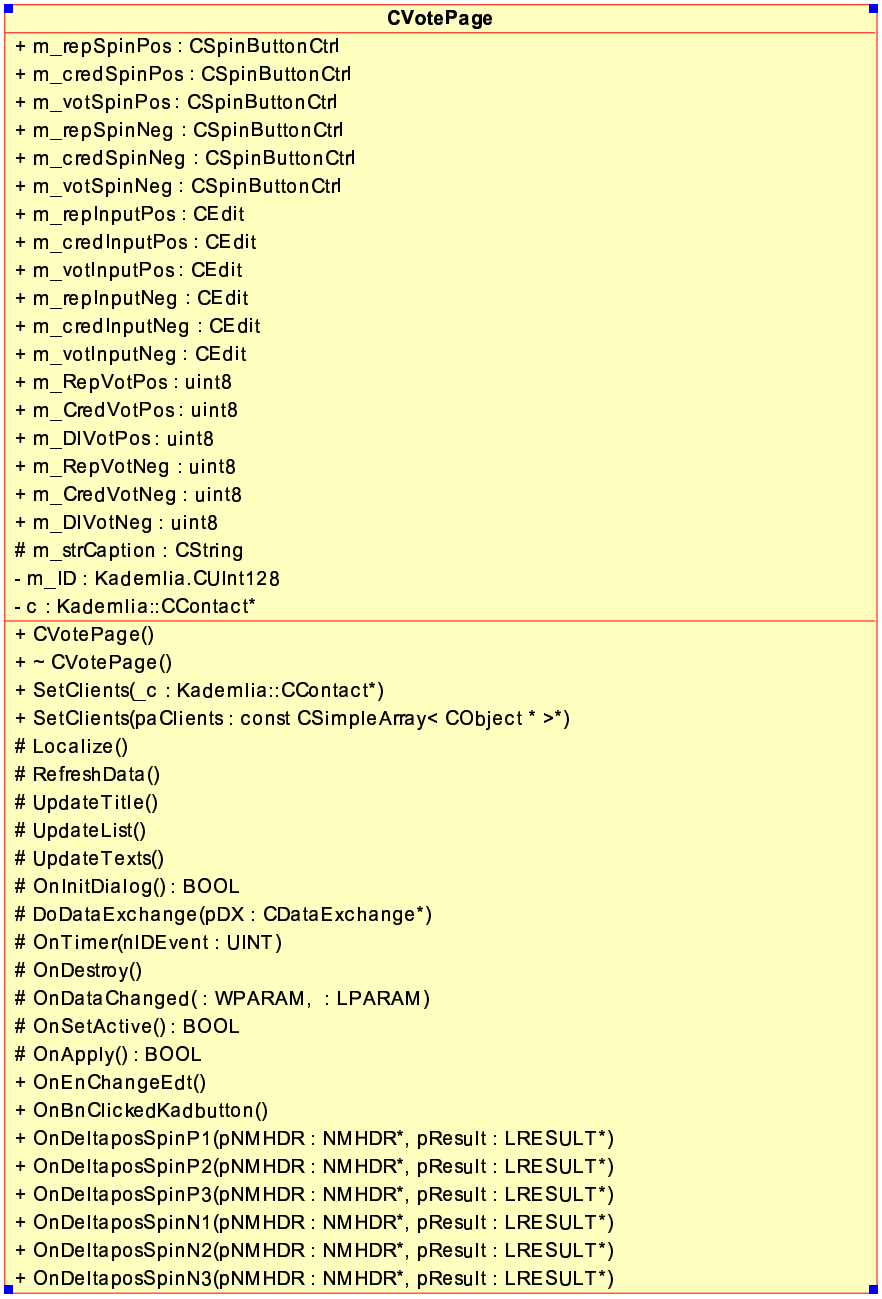


Figure 6.16: The class implementing the vote dialog

variables to the entered results. After data is consistent, the interface reflects the updated global trust value by refreshing the appropriate messages via the function `UpdateText()`.

The click handler connects the `IDC_KADBUTTON` with the function `OnBnClickedKadbutton`. This function's purpose is to call the `reputationSearch()` with the classes `m_ID` property. Various `ON_NOTIFY` functions follow. They connect the SpinButtons to their associated edit fields.

Most previously mentioned functions will be called in the `OnInitDialog()` method of the class. It sets up variables by manually invoking DDX, sets up the lists calling `UpdateLists()`, editboxes and text labels via a call to `UpdateText()`.

6.4 Security Considerations

This subchapter will try to come up with common attacks to this reputation system and its possibilities to resist these attacks.

For an introduction the features that are considered to enhance the security of this system will briefly be summed up. The basic design of Kademlia comes with a feature rich caching mechanism. While this was designed to enhance the robustness and scalability of the network it serves also as a mechanism to circumvent malicious clients. As it is not certain that every client hits a malicious user, the impact the fraud could have on the network is limited. Cryptographic methods are used to protect data, which eliminates the possibility of "tampering" values. The data itself includes the publisher and a timestamp, disabling the possibility to return valid but not updated votes. Clients have already built in features of a simple private key infrastructure. Possible false votes and bogus storage peers are measured within the client.

Client IDs are currently freely choosen. Therefore, impersonation is a problem. This problem, is however, considered structural and out of the scope of this work. The problem can be solved in various ways ranging from a registration process for users, or adding an authentication layer in the protocol basing on cryptographic information. This work relies on cryptographic information to secure its information. This spots the possibility for an attack by circumventing the caching feature.

Information is published to the *k* closest host of a network. An attacker could insert that number of hosts into the network providing the nearest matching clients himself. The client will hit a bogus client and retrieve malicious reputation information that could influence its reputation decision. Looking at the positives the client could hit a dynamical created cache on the network not under control of these peers. Furthermore a list of clients, that misuse their storage peer position is used. If the client is blacklisted already another client could be connected, or if all caches are blacklisted try to work on the data given by the peer. This data can be tampered in multiple ways. A first step is to match the list of voters against the clients own "reputation of voters" list. This could filter some of the worst voters. With the help of this list one could also gain knowledge about which peer should be "advertised" to the client. Therefore, one could decide not to use this peer. A common technique is "clique detection" that could be used when there is no previous knowledge on voters. We therefore refer to [DdVPS03]. Analyzing the data in that way could also help to detect the "pushed" client. If after applying these techniques, it is not possible to come to a decision, the network is under an attack and therefore no transaction with unknown clients should be carried out at all. Even if the reputation system is in that scenario not able to assist the users decision it can make the user aware of a high risk of transactions and warn him to handle transactions with extreme care. Therefore, such a system will not be totally useless even under worst case scenario.

Another possible problem that springs to mind when automatically receiving public keys from other hosts for verification purposes is the following scenario. Let b be a malicious client that tries to denial of service the network by creating an enormous traffic. This could be accomplished by exploiting the need to collect public keys from unknown hosts. While we agreed to see this capability as "built in" it is a problem. If *b* would be the *storage peer* of a client *c* and a client *a* would try to collect reputation information about *c*, *b* could produce a large number of votes from potentially unknown - or even nonexistent - clients. *a* would need to contact every peer listed in the data set given by *b* to retrieve the public key. This would create a lot of load all over the network.

There are several possible “workarounds” for this problem. The first is to limit the number of entries to a constant value. The client would only request a certain amount of votes. While this would limit the impact of this fraud it would certainly harm the reputation system, as it is open to the *storage peer* which votes to return as result.

Sybil Attacks In subchapter 3.4 a technique called the “Sybil” attack was analyzed. The author claims that a network will always be vulnerable, as long as an attacker can create multiple identities. Most systems are indeed vulnerable to this treat, but the impact of these attacks is overestimated. The already presented trust implementation of *Samarati* [DdVPS03] and this work share a common ground in the handling it. Both solutions require every “personality” to have a unique IP address. An attacker therefore needs to hold enough resources, to fool the system, which raises the barrier for a successfull attack immense.

7 Discussion

In the course of this work, we succeeded in creating a scalable distributed system for reputation sharing and thus created a trust system. We serve *provision trust* in the hope to raise the level of security on distributed networks by applying methods taken from real life. For building an abstraction, that closely resembles reality, a short survey of the sociological surroundings of trust was made. Furthermore, a variety of approaches on the topic of trust were taken into account and analyzed on the basis of current research on trust.

Initially, cryptographic solutions were presented, providing *identity trust,* which was identified to be the basis for all further trust research. Central reputation systems were then analyzed to get an overview of current research on trust metrics, policies and credentials. These were superseded by the distributed reputation systems, which were presented in the form of the Gnutella based system P2PRep by *Samarati et al.*

We started from the P2PRep system and tried to remain compatibility with the given semantics. The adaption the the Kademlia architecture, however, changed the underlying system dramatically. P2PRep relied on the Gnutella Protocol, which does not employ any kind of distributed hash table. Our architecture *is* DHT based and we therefore had to make different adaptions.

P2PRep did not use any kind of centralized storage therefore, the polling phase would create an enormous network overhead for every request. Our architecture solves this by using k central nodes per voted clients, that collect all votes, that should be included in a poll reply. The costs for just one run of the poll are equal in terms of messages, as all peers need to push their poll to the collecting node, but on a second poll only a request and its response are transmitted.

Moreover, the problem of the limited horizon is solved with the chosen topology. All clients send their votes to a set of nodes, that collect them in order to be able to serve votes to a requester. The horizon itself is an interesting question as well: a central storage concept provides an easy way to pollute the public measurement of trust by adding bad votes. Instead of trying to get into the neighborhood of the attacked client, as in the Gnutella system, the attacker can create a large set of votes and publish it to the responsible client. In other words the attacker does no longer need to be “near” the attacked client. However, as every major P2P architecture, including the Gnutella network, switches to distributed hash tables and attack scenarios are possible in both models, this is not seen as a critical downside of our approach.

With the new scalable search algorithm, serving in $logn$ requests, with n as number of nodes in the network, we created a scalable alternative, even when running on a large scale network. Recall, that the number of neighbours in the Gnutella network is up to 10. In contrast, in order to search a network with 60 000 000 nodes the search algorithm takes 7,778 hops, to find the responsible node.

P2PRep uses, in contrast to our work, even encryption of all data. Our work does not require this, as we rely on the integrity and authenticity features of the encryption solution. We do not use the privacy feature, because of different network topologies. In P2PRep a client encrypts the date of his `PollReply` with a public key, that is provided by the `Poll` message. The reply will then be routed back to the poller via all nodes between the responding and polling node. A malicious node that should route the packet, could decide to drop the packet if the contained vote does not suit its needs. Therefore traffic is hidden from all clients between start and endpoint of the route. In Kademlia, however, there is only direct connection between a poller and the node storing votes. This is why a privacy feature is not needed.

Of course during this period, we faced a number of problems, which will be described in the following. Most of our considerations were driven by the overhead such a network employs, when coming to a trust

decision. We believe, that a large scale adaption is only possible, if the footprint in number of messages is small. As previously described, we needed to make adaptions to the P2PRep systems to reflect our needs. We chose to include no optional verification phase for votes, as the digital signature proves, the integrity of votes. Furthermore, the node storing the votes already checks for the IP address of the voter. If a voter tries to leave a vote for an IP address that does not match his own, he will be blocked.

It has also been discussed, to include the public key of the voter in the vote, to avoid, the already mentioned need for exchanging public keys. This was drawn down due to the fact that it would be possible for the node, that is storing votes, to execute large scale frauds. It was chosen, to rely on the public key signature, which protects the system from this treat, but requires a number of message exchanges. This decision added flexibility and even security to our work. By using the PKI of the eMule client users bound their reputation to the built in credits system of eMule. A user could not abandon its Kademlia ID - and so his bad reputation - easily, as he would also lose the associated credits, that effectively identify his rank in other clients waiting queues. But other solutions are possible as well. Further keys retrieved from existing PKIs, as the web of trust, or public key servers could add a better foundation of trust to existing solutions.

Another interesting feature would have been the publication of the reputation as voter. By retrieving the attributes separately, a more elaborate policy would have been possible, with more fine grained control over the mechanisms leading to trust relations could have evolved. We chose to not include this feature, as it would have involved another $k\ n$ messages for n voters, to different storage nodes. For enabling more control over relations we talk in the next chapter about the concept of risk, which will enable some of these benefits.

Further problems were implementation specific. As Microsoft migrates its compiler to become fully ISO C99 standard compatible, there were issues with the compilation and runtime of the program, that needed to be addressed. The code quality of parts of the eMule source did not allow us any deeper integration into the system. At last, the testing of the provided features needed some additions to the code base, as we found no default ways to debug a large scale distributed system locally.

8 Conclusion

With this work, we have created an established basis for further research in all main directions of trust research. While the presented trust solution is both scalable, secure and fully decentral, further research could extend the system in some areas. We want to give some pointers in the chapter.

Integration One problem that could easily addressed is finding more credentials. Both automated as half automated credentials would provide a set of additional information, that could be used to define tighter trust policies. These would include measurements of speed, registered port scans from other nodes or manual interactions with the peer, like whether a chat between both users had been initiated previously. By using more and more provided data, techniques from common intrusion detection systems like snort[1] or their information could be used. This would also serve as a testbed for data mining techniques in the surrounding of computer science.

Sanctioning Strategy The policy system presented in this thesis is currently using a pessimistic approach. This strategy is intuitively safe. However game theory draws an interesting picture. In chapter 2.3 we transitioned from sociology to computer science using the "prisoners dilemma". Although the only Nash equilibrium in the game is the "both cheat" situation, research has proven that a optimal strategy is the "nice" strategy. This strategy returns to playing fair, after sanctioning a cheating player. This is an interesting scenario for a P2P systems, that should receive attention. Currently after settling a negative trust decision, it is possible, that a sanctioned client will not be distrusted, as enough other clients still believe in its trustworthiness. But if a majority of voters certifies the client as not trustworthy, how could the client regain its trust? In the current implementation this was considered out of scope, since the client gains its trustworthiness, as the voter leaves the system. After the republish interval of his votes has ended, they will be discarded. It is also possible that another client does an interaction, although the peer is not trusted. The client has a chance to gather a positive recommendation. However in a work based on this thesis a "expiry" could be introduced, that invalidates votes after a period of time, resetting the trust relation back to the initial state. It would be interesting to see, how this approach would affect the trust relations on the network, as a whole.

Credibility and Reliability Our current approach is designed to enable users to build a trust decision, when they face peers, they have never had interaction before. It would also be interesting, on how to combine the information, when there has been previous interaction between the peers. How could a possible trust decision be computed? Should the users own vote be included with a double weight? Further how should the credibility of voters be computed? If they are trusted as download source, should this count more as a couple of bad votes? These are definitively questions that should be answered on the path to a fully automated system, that operates without user interaction, on the recommendations a trust system produces.

Trust Model and Policy Another interesting work in this area, would be the switch from reliability trust, to decision trust, as defined in chapter 2.1. This change would introduce a user specific risk concept. In the current form of our system, there is no opportunity to influence the decision of the policy, that controls the trust relations. However for building fully automated trust systems, it would be feasible, to add parameters to the computation, that allow influence on the risk factor. Clients, that accept trust relations, even at high risk, or change the risk factor depending on a specific context come to mind. The idea is similar to the approach taken by Blaze, described in chapter 3.1.5, that supports fully programmable policies, depending on the context. A P2P system could provide these concepts, based on the requested usage. Just like the web browser, that accepts

[1] http://www.snort.org

expired SSL keys in a given interval, rejecting them from mission critical sites, like a bank, a P2P system used for downloading a PGP key, should adapt its risk tolerance to low. In contrast when retrieving information in "secure" formats, e.g. textfiles could accept even a high risk, as these files do not contain executable code and are usually small in size. So downloading and even executing them is acceptable, even at high risk.

Credentials A completely other set of credentials becomes mandatory, when porting this application of trust in another scenario. For a applied scenario, we will model a trust system for a real world scenario.

The well known CERT institute from Genve possesses the worlds largest particle accelerator. When doing experiments a incredible amount of data has to be handled. For that purpose there exists a escalation strategy, where the data is distributed to a couple of large scale electronic data processing centres (EDPC). The analysis and storage of the data is then once more escalated from the large scale EDPC, to smaller centres. For example one such smaller center is the "Leibnitz Rechenzentrum" in Munich. Of course when analyzing that strategy, we will find out, that the topology is a tree, with the data producer as root and smaller workstation, that actually do the processing of data as leafs.

Of course these scientific networks also face the problem of trust relations. At the present time, all nodes are chained with cryptographic keys, that only provide *identity trust.* However a strategy to check the validity of the given services, that would provide *provision trust* is not developed. A possible attack scenario that would be caught by applying that kind of trust would be an attack on the data, or the analysis of it. The computations could be altered by an attacker and as long as the cryptographic key chain is not altered, there would be no possibility to detect this.

So what would a trust model, that supports a use case, like the above mentioned, look like like? In the course of this work we were using a reputation system, for the evaluation of trust relations. As we now have a hierarchical trust model, with an ultimately trusted root, we need to adapt our credentials. We no longer need a publicly available measurement of trust, but need a structured measurement of trust. For the leaf nodes in the tree, the trust relation between their parent nodes is not interesting. The only trust chain, that is really interesting is the "top-down" approach, as the information "provider" needs to be sure that the next computer, on the path to the leafs is trusted. This approach is already resembled and well tested, in the cryptographic identity trust checking mechanisms. When passing data to the next node, the public key of that machine is checked. After the identity has been proven, the further distribution is done by the "lower" node.

Checking the credentials of provision trust would mean that any given node could pass data down to *two* different machines. These machines would do storage, or analysis on the given sets of data. By doing so the "parent node" would waste some storage or CPU cycles, but could compare the results of both operations and compare them for differences.

Finally Due to the size and rapid development of both the P2P and the trust system area this work can only provide a snapshot of the current state of the art. The future will hopefully bring a wide spread usage of these techniques. The problems of central systems, like the PKI systems, in use today have been spectacular demonstrated, by a fraud described in *http://www.heise.de/newsticker/meldung/69598*. A malicious user has been able to provide a SSL certificate, signed by GeoTrust, that identified him as the Mountain America bank. As GeoTrust is a well known Certificate Authority, this marks a new generation of frauds and identifies some of the weaknesses of relying on a centralized Identity Trust system.

Bibliography

[AC05] Inc. Apple Computer. Security concepts, 2005. http://developer.apple.com/documentation/Security/Conceptual/Security_Overview/Concepts/chapter_3_section_6.html.

[AJ05] Colin Boyd Audun Jøsang, Roslan Ismail. A survey of trust and reputation systems for online service provision. In *Decision Support Systems*, 2005.

[BFK99] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. Keynote: Trust management for public-key infrastructures (position paper). *Lecture Notes in Computer Science*, 1550:59–63, 1999.

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. Technical Report 96-17, 28, 1996.

[BG99] Dan Brickley and R.V. Guha. Resource description framework (rdf) schema specification. Technical Report http://www.w3.org/TR/rdf-schema/, 1999.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[CCI88a] CCITT. The directory: Authentification framework. Draft Recommendation X.509, 1988. Version 7.

[CCI88b] CCITT. The directory: Overview of concepts, models and services. Draft Recommendation X.500, 1988. Version 7.

[Cli01] Clip2. The gnutella protocol specification v0.4, 2001. http://www.clip2.com/GnutellaProtocol04.pdf.

[DdVPS03] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Managing and sharing servents' reputations in p2p systems. *IEEE Transactions on Data and Knowledge Engineering*, 15(4):840–854, July/August 2003.

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

[Dou02] John Douceur. The sybil attack. In *Proceedings of the 1st International Peer To Peer Systems Workshop (IPTPS 2002)*, March 2002. http://www.cs.rice.edu/Conferences/IPTPS02/101.pdf.

[Dum02] Edd Dumbill. Finding friends with xml and rdf, June 2002.

[Fou05a] Wikimedia Foundation. Social networks, 2005. http://en.wikipedia.org/wiki/Social_networks.

[Fou05b] Wikimedia Foundation. Social networks, 2005. http://en.wikipedia.org/wiki/PageRank.

[Gam88] Diego Gambetta. *Can We Trust Trust?*, chapter 13, pages 213–237. Basil Blackwell, 1988. Reprinted in electronic edition from Department of Sociology, University of Oxford.

[Gen86] J. V Genest, C. Zidek. Combining probability distributions: A critique and an annotated bibliography. *Statistical Science*, 1:114–148, 1986.

[Goo04] Google. Google technology, 2004. http://www.google.com/ technology/.

[Han00] Rob Handfield. What does it mean to trust?, 2000. http://scrc.ncsu.edu/public/DIRECTOR/dir110503.html.

[Hof85] Douglas Hofstadter. *Metamagical Themas: questing for the essence of mind and pattern*, chapter 29. Bantam Dell Pub Group, 1985.

[Inc05] RSA Security Inc. Crypto faq, 2005. http://www.rsasecurity.com/rsalabs/node.asp?id=2214.

[Kol99] Peter Kollok. The production of trust in online markets. In *Advances in Group Processes (Vol. 16)*. JAI Press, 1999. http://www.sscnet.ucla.edu/soc/faculty/kollock/papers/ online_trust.htm.

[KR97] Rohit Khare and Adam Rifkin. Weaving a web of trust. *World Wide Web J.*, 2(3):77–112, 1997.

[KS98] B. Kaliski and J. Staddon. Pkcs #1: Rsa cryptography specifications version 2.0. RFC 2437, oct 1998. ftp://ftp.isi.edu/in-notes/rfc2437.txt.

[KSGM] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks.

[Lev00a] Raph Levien. Mission statement, 2000. http://www.advogato.org/ mission.html.

[Lev00b] Raph Levien. Mission statement, 2000. http://www.advogato.org/ trust-metric.html.

[Lev04] Raph Levien. Attack resistant trust metrics, 2004.

[MC96] D. Harisson McKnight and Norman L. Chervany. The meanings of trust. Technical Report 94-04, Carlson School of Management, University of Minnesota, 1996. http://misrc.umn.edu/wpaper/WorkingPapers/9604.pdf.

[MKKB01] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[MM02] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[Pea88] J. Pearl. Probabilistic reasoning in intelligent systems: Networks of plausible interference, 1988.

[Pro06] Emule Project. Credit system, 2006. http://www.emule-project.net/home/perl/help.cgi?l=1&topic_id=134.

[RKZF00] Paul Resnick, Ko Kuwabara, Richard Zeckhauser, and Eric Friedman. Reputation systems. *Commun. ACM*, 43(12):45–48, 2000.

[RSA77] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, 1977. http://theory.lcs.mit.edu/ rivest/rsapaper.pdf.

[Wat76] Paul Watzlawick. *How Real is Real?* Vintage, 1976.