Webs of Trust
in
Distributed Environments

Bringing Trust to Email Communication

Bachelor Thesis

Marko Brosowski

Supervisors:

Prof. Dr. techn. Wolfgang Nejdl
Prof. Dr. Nicola Henze
Dipl.-Ing. Daniel Olmedilla

Information Systems Institute, Knowledge Based Systems
Department of Computer Science, University of Hanover

I’d like to thank
Prof. Dr. techn. Wolfgang Nejdl and Dipl.-Ing. Daniel Olmedilla
for their great support.

Also I’d like to thank
Dr.-Ing. Jörg Diederich and Dipl.-Ing. Paul-Alexandru Chirita
for their help with PageRank, Appleseed and the thesis in general.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, 6. September 2004

. . . . . .

Marko Brosowski

Eingegangen am (Datum/Stempel): _____

Contents

Chapter 1

Introduction and Motivation

Email communication is an important part of our life. It is fast and cheap, but also often abused by people who sending unwanted messages (Spam) around and filling our mailboxes.

This thesis will show a way to express and infer trust, based on email-addresses and how this can be used to classify and filter unwanted messages.

First I give an example of trust and reputation based on certificates. Then I show widely used trust and reputations algorithms. Finally I introduce Mailrank: the Mailrank algorithm, which is still subject to change and the Mailrank server software, which was my major task within this thesis.

1.1 Trust and Reputation based on Certificates

Certificates are based on RSA public key cryptography. This asymmetric cipher method was developed in 1977 by Ron Rivest, Adi Shamir and Len Adleman. A certificate is nothing else than the public key with additional information such as issuer, validity, entryption strengh et cetera. According to RSA a private key always belongs to a certificate.
A key introduction of public key cryptography can be found in [8].

Certificates provide a secure way to authenticate people. Authentication provides the verification of an identity.

For instance when a user checks his[1] email, he must send a password to the server to confirm his identity. Then the user is authenticated and can get his emails.

In general authentication is done with credentials. Passwords are credentials as well as certificates.

[1] In this thesis I will use the form 'he'. Of course this also means: 'she'

Today two standards for Public Key Cryptography are in use: X.509 [9] and GPG/PGP ([14],[15]).

1.1.1 X.509 Standard

X.509 is the recommended standard for Public Key Infrastructures (PKI). It's implemented in all browsers, email clients and server software dealing with PKI.

The main concept of X.509 is a hierarchical tree structure, where the root represents an issuer of a certificate called the root certificate authority (root CA). Common root CAs are Verisign (http://www.verisign.com), Thawte (http://www.thawte.com) or Trustcenter (http://www.trustcenter.de). Figure 1.1 shows the structure of a PKI.

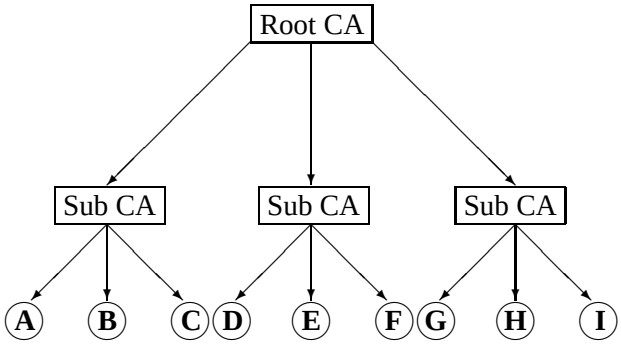


Figure 1.1: Example of a X.509 Public Key Infrastructure

Trust is expressed with accepting the root CA certificate as trusted. That also means, because of the tree structure a user automatically trusts all certificates issued by this certificate authority.

1.1.2 The PGP/GPG Standard

A big flaw of X.509 is if somehow the root CA is compromised, none of the issued certificates can be trusted. Also the tree structure is not much flexible.
Because of these big disadvantages, in 1991 a new approach was developed.

Instead of a hierarchical tree structure, in PGP/GPG[2] a peer-to-peer based trust is used. This means every user is his own root CA. Trust is expressed

[2]The main difference between GPG and PGP is that GPG is free software under the Gnu Public License (GPL), while PGP is a commercial product.

by signing the public key of a user with the own private key (direct trust) or by inference of trust from the local key database (indirect trust). Table 1.1 shows the different levels[3] for signing, while table 1.2 shows the constraints for inference of trust.

Trustlevel	Description
Unknown	There is nothing known of the capabilities of the user
None	The user is not trustworthy
Marginal	The owner understands the implication of key signing and properly validates keys before signing them
Full	The owner has an excellent understanding of key signing, and his signature on a key would be as good as your own
Ultimate	Only used for own keys. This trust level determines a key that is similar to a root CA certificate in X.509

Table 1.1: Trustlevels in PGP/GPG

In X.509 inference of trust is simple. If someone trusts a root CA he automatically trusts all of the users of this root CA.
In PGP/GPG every user is his own root CA and trust is inferred based on the trustlevels described in figure 1.1 and the rules shown in figure 1.2. A trust value in GPG/PGP is either 0 or 1, meaning the key is evaluated as valid (1) or not valid (0). A key is evaluated as valid if a signing and the pathlength rule is true.

Signing	Pathlength
The key is personally signed by the user that evaluates the validity (direct trust)	The path from the own key to the key whose validity is evaluated is exactly or shorter than a certain number (configurable)
The key has been signed by a number (configurable) of fully trusted valid key (indirect trust)	
The key has been signed by a number (configurable) of marginal trusted valid keys (indirect trust)	

Table 1.2: Inferrence of Trust in PGP/GPG

[3]The levels "unknown" and "ultimate" are not available in PGP

1.2 Conclusion

The peer-to-peer based approach of PGP/GPG is much more flexible than the hierarchical one of X.509. A much more differentiating use of trust values is possible and fits well in todays peer-to-peer infrastructures like PeerTrust [10] and EDUTELLA [11].

Though the current implementations of PGP/GPG include no propagation of trust values, so this is local only, but researchers have recently begun to develop algorithms for trust- and reputation propagation.

Chapter 2

Trust and Reputation Algorithms

In this chapter I first define trust and reputation. Then I give an overview of well known trust and reputation algorithms.

2.1 Definitions

A definition of trust and reputation is difficult because of the subtle nature and complexity. Therefore most of the authors including Guha [1], Ziegler [5] and Kamvar [3] omit any definition.

2.1.1 Trust

In this thesis the trust definition from Mui et al., 2001 [6] is used: "[trust is] a subjective expectation an agent has about another's future behavior based on the history of their encounters".

2.1.2 Reputation

Mui et al. 2001 [6] also gave a definition for reputation: "[reputation is a] perception that an agent creates through past actions about its intentions and norms". While norms are defined in Ostrom, 1998 [7]: "[norms are] heuristics that individuals adopt from a moral perspective, in that these are the kind of actions they wish to follow in living their life."

2.2 Overview

2.2.1 Propagation of Trust and Distrust

Guha et al. [1] found a way to express and propagate not only trust but also distrust in a network of people. They give eBay (http://www.ebay.com) and Epinions (http://www.epinions.com) as an example why distrust is as important as trust. The price of an item or opinion about a product can be forged by a group of users for their own benefit.

Computation of trust values in this paper is done in four steps:

1. Calculating a matrix of beliefs $C_{B,\alpha}$ consisting of four matrices, weighted with a vector α
2. Choosing an algorithm for distrust propagation
3. Choosing an algorithm for iterative propagation
4. Rounding to interpret the result values

1. **Calculating** $C_{B,\alpha}$
   The algorithms used in their approach are based on matrices and matrix operations. A matrix of trust (T), that models the trust values between two persons i and j (value between 0 and 1), a matrix of distrust (D) and a matrix of beliefs (B), that represents trust ($B = T$), distrust ($B = D$) or a combination ($B = T - D$). With these matrices they defined four different types of trust propagation. These are called atomic propagations, because they represent one step in propagating trust. (Table 2.1)

   With these atomic propagations they defined a combined matrix $C_{B,\alpha}$ with $\alpha = (\alpha_1, \alpha_2, \alpha_3, \alpha_4)$ the weight vector as follows:

$$C_{B,\alpha} = \alpha_1 B + \alpha_2 B^T B + \alpha_3 B^T + \alpha_4 B B^T$$

2. **Algorithm for distrust propagation**
   Considering $P^{(k)}$ is the matrix whose entries represent the beliefs after k propagation steps, they give three ways to propagate distrust:

   Trust Only: Ignoring distrust completely. Then the matrix of beliefs B and $P^{(k)}$ is as follows:

$$B = T,\ P^{(k)} = C_{B,\alpha}$$

Atomic propagation	Operator	Description
Direct propagation	B	If A trusts B, and B trusts C, A automatically trusts C. This means A have trust in a friend of a friend (FOAF) (figure 2.1(a))
Co-citation	$B^T B$	If A trusts B,C and D trusts C, this implies D trusts C. In other words: D trusts people's friends with trusting D's friends. (figure 2.1(b))
Transpose trust	B^T	This is similar to direct propagation, but the other way around. If A trusts B and C trusts B, then B automatically trusts A. That means C trusts people, that trusting C's friends. (figure 2.1(c))
Trust coupling	BB^T	This is also similar to co-citation, but the graph direction is different. If A,C trust B, C trusts B and D trusts A, this implies D trusts C. In other words: D trusts people that trusts D's friend's friends. (figure 2.1(d))

Table 2.1: Atomic propagations in "Trust and Distrust" [1]

One-Step-Distrust: If a person A distrusts another person B, then A ignores all judgements made by B, so distrust is expressed in only the first step of propagation.

$$B = T,\ P^{(k)} = C_{B,\alpha} \cdot (T - D)$$

Propagated Distrust: Propagation of trust and distrust together

$$B = T - D,\ P^{(k)} = C_{B,\alpha}$$

3. **Algorithm for iterative propagation** After choosing the propagation algorithm the final belief matrix F can be computed with two algorithms:

   Eigenvalue Propagation (EIG): Let K be the number of iterations, then the final matrix is given by:

   $$F = P^{(K)}$$

   Weighted Linear Combination (WLC): The idea behind WLC is to weight every iteration with a parameter γ, so that:

   $$F = \sum_{k=1}^{K} \gamma^k \cdot P^{(k)}$$

4. **Rounding of results** The final step is to interpret every element of F as either trust or distrust. This is done with one of the following rounding algorithms:

(a) Direct Propagation (b) Co-Citation

(c) Transpose Trust (d) Trust Coupling

Figure 2.1: Atomic Propagation

Global Rounding: Let τ be a chosen threshold based on the overall fractions of trust and distrust in the initial input. Then i trusts j if and only if the element F_{ij} of the matrix F is within the top τ fraction of entries of the row vector F_i.
Example: Let

$$F_i = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 3 \\ 4 \end{pmatrix} \text{ the } i\text{-th row vector of } F = \begin{pmatrix} & 3 & 1 & 3 & \\ & 8 & 2 & 5 & \\ \cdots & 8 & 2 & 2 & \cdots \\ & 5 & 3 & 0 & \\ & 1 & 4 & 4 & \end{pmatrix}$$

$\tau = 3$ and $F_{ij} = 2$ the second element of F_i. The element $F_{ij} = 2$ is in the τ top fraction of the vector F_i under the $<$ ordering (F_{ij} is on position 2), therefore i trusts j.

Local Rounding: This rounding is similar to global rounding, but the threshold τ is chosen based on the relative fraction of trust vs. distrust judgments made by i.

Majority Rounding: Let J be a set of users that i had already judged with either trust or distrust. J is ordered according to the structure of F_i, that means J basically becomes a vector and every J_k is the trust or distrust judgment of F_{ik}, that is the trust/distrust of i for k.
Finally the decision whether trust or distrust j is made based on the majority of trust/distrust judgments in vector J.
Example: Let

$$F_i = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 3 \\ 4 \end{pmatrix} \text{ the } i\text{-th row vector of F and } \quad J = \begin{pmatrix} - \\ j \\ + \\ + \\ + \end{pmatrix}$$

This finally ends up with trusting j (the $+$ are in majority).

2.2.2 Inferring Reputation on the Semantic Web

Golbeck et al. [2] present an algorithm for inference of reputation based on directed graphs where the nodes representing users and every edge is provided with a reputation rating. The algorithm is designed to make local reputation ratings and make them available for other nodes. Reputation values only use a {0,1} scale and ratings are rounded up to 1 for [0.5-1] and down to 0 for values less than 0.5. 1 means good reputation while 0 is considered as bad reputation.

Reputation from a node A to a node C is inferred by A polling all his neighbors to which A has a good reputation rating, asking for a rating of C. Every neighbor will answer the query, but nodes with bad reputation rating will be ignored by A. Then A takes the received values, computes the average of them and round the value like described above. Figure 2.2 shows an example, where reputation from A to C via B is inferred.

2.2.3 The EigenTrust Algorithm for Reputation Management in P2P Networks

EigenTrust developed by Kamvar et al. [3] is a reputation algorithm for peer-to-peer (P2P) environments. It is based on transitive trust. That means a peer i will trust another peer j with it's opinion about other peers as well as j itself.

In general they define: "[...] the global reputation of each peer i is given by the local trust values assigned to peer i by other peers, weighted by the global reputations of the assigning peers."

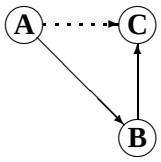


Figure 2.2: Reputation from a node A to a node C via B in [2]

Background: Considering s_{ij} as the sum of ratings for transactions between a peer i and a peer j, each rated with either 1 for a good transaction or -1 for a bad result. Normalizing these results gives a trust value of i for j at each peer:

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

To aggregate trust values for a peer k, a peer i will have to ask all well known peers about their opinion of k. Then the vector of transitive trust is given by:

$$t_{ik} = \sum_j c_{ij} c_{jk}$$

A better way to write this down is to define a matrix C with all values c_{ij} and a vector $\vec{t_i}$ that includes all t_{ik}. Then it follows:

$$\vec{t_i} = C^T \vec{c_i}$$

with c_i normalized local trust vector for peer i. The goal for a peer i is to have a complete view of the network. After querying the well known peers about an opinion, i will ask the 'friends' of these peers and so on. For the equation this means:

$$\vec{t_i} = (C^T)^n \vec{c_i}$$

Finally after n iterations, i will have a complete view of the network and the global trust vector ($\vec{t}$) as well as all the other peers.

Now the algorithm is clear, but the way the vector $\vec{t}$ is computed can be different. They give three possible ways of computation, starting with a basic EigenTrust algorithm, extending it to a distributed variant and finally describing secure EigenTrust, which correct some flaws.

Basic EigenTrust: For this basic variant the P2P aspect of the network is completely ignored and the computation takes place in a central system.

This leads to:

$$\vec{t}^{(0)} = \vec{e};$$
repeat
$$\vec{t}^{(k+1)} = C^T \vec{t}^{(k)};$$
$$\delta = ||\vec{t}^{(k+1)} - \vec{t}^{(k)}||;$$
until $\delta < \varepsilon;$

Several issues are not considered by this algorithm:

1. *Pre-trusted peers:* In a P2P network some peers should have higher reputation ratings from the beginning than all the others. This is done by adding a start vector $\vec{p}$, so that $\vec{t} = (C^T)^n \vec{p}$ converges faster than $\vec{t} = (C^T)^n \vec{e}$.

2. *Inactive peers:* They redefine c_{ij} to incorporate this:

$$c_{ij} = \begin{cases} \frac{max(s_{ij},0)}{\sum_j max(s_{ij},0)} & \text{if } \sum_j = max(s_{i,j},0) \neq 0 \\ p_j & \text{otherwise} \end{cases}$$

   while p_j is the i-th component of start vector $\vec{p}$.

3. *Malicious Collectives:* A group of peers (G) can easily subvert the P2P system by giving all members of G high and all other peers low local trust values. For omitting that a factor $a < 1$ is added to the equation of $\vec{t}^{(k+1)}$ and the final basic EigenTrust algorithm is:

$$\vec{t}^{(0)} = \vec{e};$$
repeat
$$\vec{t}^{(k+1)} = C^T \vec{t}^{(k)};$$
$$\vec{t}^{(k+1)} = (1-a) C^T \vec{t}^{(k)} + a\vec{p}$$
$$\delta = ||\vec{t}^{(k+1)} - \vec{t}^{(k)}||;$$
until $\delta < \varepsilon;$

Distributed EigenTrust: The next step is to compute the trust vector not central but distributed. Peer i for example can simply compute it's own global trust value with:

$$\vec{t_i}^{(k+1)} = (1-a)(c_{1i} t_1^{(k)} + \ldots + c_{ni} t_n^{(k)}) + a p_i$$

which is the i-th component of $\vec{t}^{(k+1)}$.
Note: Only pre-trusted peers need to know their p_i.

Secure EigenTrust: With distributed computation some security considerations come into play. First, malicious peers can easily report higher trust values for themselves by manipulating the computation. Second, these peers can also send wrong results when computing other peers trust values. The solution is to use more than one peer to compute a trust value. These peers are called score managers. Each peer has a number M of score managers and the direct position of these is veiled by using a distributed hash table (DHT). A discussion about DHTs is beyond the scope of this thesis, but good references are [12] and [13].

With these extensions, the final secure EigenTrust algorithm can be defined.

Let M be the number of score managers for each peer, $h_0, h_1, \ldots, h_{M-1}$ a number of one-way hash functions and pos_i the position of a peer i in the hash space.

Every peer can act as a score manager. The indexes of the peers whose trust value computation is covered in a score manager is referred as D_i, also called the set of daughter peers.

A score manager i also maintains an opinion vector c_d^i for all of his daughter peers d ($d \in D$).

Furthermore i maintains a set of peers A_d^i that downloaded files from it's daughter peers d. i will get trust assessments for it's daughter peers d from this peers. Also, a set of peers B_d^i is stored which determines the peers, i's daughter peers d downloaded files from.

The finally secure EigenTrust algorithm is depicted in figure 2.3.

foreach peer i **do**

Submit local trust values $\vec{c_i}$ to all score managers at positions $h_m(pos_i), m = 1 \ldots M-1$;

Collect local trust values $\vec{c_d}$ and sets of acquaintances B_d^i of daughter peers $d \in D_i$;

Submit daughter d's local trust values c_{dj} to score managers $h_m(pos_d), m = 1 \ldots M-1, \forall j \in B_d^i$;

Collect acquaintances A_d^i of daughter peers;

foreach *daughter peer* $d \in D_i$ **do**

Query all peers $j \in A_d^i$ for $c_{jd} p_j$;

repeat

Compute $t_d^{(k+1)} = (1-a)(c_{1d}t_1^{(k)} + c_{2d}t_2^{(k)} + \ldots + c_{nd}t_n^{(k)}) + ap_d$;

Send $c_{dj}t_d^{(k+1)}$ to all peers $j \in B_d^i$;

Wait for all peers $j \in A_d^i$ to return $c_{jd}t_j^{(k+1)}$;

until $|t_d^{(k+1)} - t_d^{(k)}| < \varepsilon$;

end

end

Figure 2.3: The Secure EigenTrust Algorithm

2.2.4 The PageRank Citation Ranking: Bringing Order to the Web

Pagerank [4] is the algorithm from Google (http://www.google.com) developed in 1998 by L. Page, S. Brin, R. Motawi and T. Winograd.

The algorithm determines the importance of web pages, also called rank. This is done by describing the Web as a directed graph, where the nodes are the web pages and the edges are links from one to another page. Then every page has links that points to it (called back links) and links that points to other web pages (forward links).

An intuitive definition of PageRank is: "a page has a high rank if the sum of the ranks of it's backlinks is high." ([4])

They give two equations for computing PageRank: A simplified version and the final version of PageRank.

The simple PageRank

Let u be a web page, F_u the set of pages u points to (forward links), B_u the set of pages that point to u (back links) and $N_u = |F_u|$ the number of links from u. c is a factor used for normalization. This makes sure that the total rank of all web pages is constant. Then the rank of a web page u is:

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v}$$

The final PageRank

The simple version of PageRank doesn't address one problem: If two web pages are linked to each other, but don't point to any other page, the algorithm will accumulate rank but never distribute this rank values. These two pages form a loop, also called rank sink. The final PageRank algorithm considers this problem.

Let A be the matrix corresponding to the web graph. If a web page u is linked to a web page v, let $A_{u,v} = \frac{1}{N_u}$, if not $A_{u,v} = 0$. Also let R be a vector over web pages, then $R = cAR$ is the eigenvector of A with eigenvalue c.

To consider the rank sink problem, a vector $E(u)$ is defined that models the behavior of a surfer, who visits a web page and after a certain time follows a link to another web page. This is called the random surfer model.

Then PageRank R' is defined as:

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u)$$

with c maximized and $||R'||_1 = 1$ ($||R'||_1$ is the L_1 norm of R')

2.2.5 Spreading Activation Models for Trust Propagation

The AppleSeed algorithm [5] introduced by Ziegler and Lausen is motivated by spreading activation strategies. These are adapted for trust propagation and local group trust metrics.

Local group trust metrics are a category of trust metrics, that addresses computation of local trust values in a centralized way, "[Here used] in their function as means to compute trust neighborhoods" ([5])

Spreading activation models are "models of retrieval from long-term memory in which activation subdivides among paths emanating from an activated mental representation" ([23]). In this context they are used to propagate trust in a network.

Spreading Activation

The basic idea of spreading activation is to propagate energy in a network, where edges connect nodes, which is basically a directed graph. This is similar to PageRank described in 2.2.4.

Additionally the edges between nodes are weighted. That means the higher the weight of an edge (x,y) connecting two nodes x and y the higher the amount of energy that flows along y. Also, the closer x to the injection source s and the more paths leading from s to x the higher amount of energy flows into x. Finally an energy flow into x has to exceed a threshold T or it's ignored. This prevents endless, marginal and negligible flow in the network.

Trust Propagation

The idea of energy flow is tailored for trust propagation, but some adaptations have to be made.

1. The energy e passed through a node x is passed without loss to its successor nodes. While using energy as a representation for rank, this means every successor node of x has the same rank as x itself. A trust decay is needed.

2. Energy that is injected in a cycle of nodes (for example three nodes a,b,c that point $a \to b \to c \to a$) and will rise the rank of this nodes to infinity. This is similar to the rank sink problem in PageRank (2.2.4).

1. **Trust decay**

   To decay trust in a chain of nodes a spreading factor d for the partitioning of the energy, respectively trust, is used.

   Let d the spreading factor with $(0 < d < 1)$ and $in(x)$ the energy that flows into a node x. Then $d \cdot in(x)$ is the energy left for the successors of x, while $(1-d) \cdot in(x)$ is the energy for x itself.

The spreading factor d can also be interpreted as the ratio between direct trust in a node x and trust in the ability of x to recommend others as trustworthy peers.

2. **Rank normalization and rank sinks**

   Appleseed uses edges that are weighted. For propagation of energy/trust that means the successors of a node x receive energy based on x's weight in the graph.

   For example: Let $e_{x\to y}$ the quantity of energy distributed along (x,y) from x to y. Then $e_{x\to y}$ depends on the relative weight of x compared to the sum of weights of all outgoing edges of x.

   $$e_{x\to y} = d \cdot in(x) \cdot \frac{W(x,y)}{\sum_{(x,s)} W(x,s)}$$

   This problem is solved by the use of backward propagation.

   Every time a computation is made, "virtual edges" from every node x to the trust source are created. These edges are weighted with full trust ($W(x,s) = 1$) and existing backlinks are overwritten. Every node is forced to trust the source s. This leads to two improvements:

   (a) Mitigating relative trust: trust distribution through backward propagation links are much fairer for successors of a node than relative trust.

   (b) Avoidance of dead ends: with backward propagation all nodes are explicitly linked to the source node s, so there are no more dead ends. This also solves the rank sink problem.

   One disadvantage is left though: For nodes close to the source s, backward links to s are favorable for their rank, while further away nodes have a disadvantage.

Improvements: *Nonlinear Trust Normalization*

Let x be a node with poor rating from y. Because of the low outdegree of y x is awarded with high overall trust rankings. A possible solution is to take the square of local trust weights:

$$e_{x\to y} = d \cdot in(x) \cdot \frac{W(x,y)^2}{\sum_{(x,s)} W(x,s)^2}$$

Finally the amount of incoming trust for a node i with predecessors p is as follows:

$$in(x) = d \cdot \sum_{(p,x)} \left(in(p) \cdot \frac{W(p,x)}{\sum_{(p,s)} W(p,s)} \right)$$

The trust rank for x is updated with:

$$trust(x) \leftarrow trust(x) + (1 - d) \cdot in(x)$$

When no new nodes have been discovered or the changes of the trust ranks are not greater than the accuracy threshold T_c the algorithm terminates.

Chapter 3

Mailrank

In this chapter I first give an introduction of Mailrank. Then I describe the Mailrank server which was my major task in this bachelor thesis.

The Mailrank algorithm is subject to investigation, so the definition is preliminary and subject to change. Intensive tests have to be made, though I'd like to present the idea of Mailrank.

3.1 The Mailrank Algorithm

PageRank as shown in 2.2.4 is generally used to compute the importance of web pages. Appleseed as shown in 2.2.5 is based on spreading activation models and used to propagate personalized trust.

These ideas are adapted in Mailrank to compute the rank of Spam email addresses and build a web of trust for email users. Instead of web pages email addresses are used as nodes. The links are directed from a user's email address towards all addresses in his address book as well as his AutoWhitelist.

3.1.1 Computation

From a users point of view the Mailrank server (see 3.2) maintains two tables: an addressbook for "good" email addresses and a Spam table, where all the Spam email addresses are stored, that were submitted by the user's Spamassassin plugin (3.2.1).

Ranking of Spam Emails

The Spam table of all users on the server is used to compute a score for a certain Spam email address. The information in the Spam table are:

- The user that submits the information (U)
- The Spam email address ($Spam$)[1]
- The score (S)
- The count of emails receive from this Spam email address (C)

According to Pagerank an email address has a high rank if the sum of the ranks of it's backlinks is high. Here the backlinks are users that submits the information. That means the Mailrank MR of a Spam email address $Spam$ is defined as:

$$MR(Spam) = c \sum_{U} \frac{MR(U)}{N_U}$$

where

$$N_U = \text{Number of all users}$$

Then this Mailrank is used to judge an email, like:

- if MR is in the top 20% of all non-Spam email addresses, add -5 to the Spam score
- if MR is in the last 20% of all non-Spam email addresses, add +10 to the Spam score

The Mailrank value is also propagated to all users using Appleseed.

This is a very preliminary version of the algorithm. The exact algorithm is subject of investigation, though I hope this version gives an idea of how Mailrank works.

[1] not the email address, but a hash of the email address is used to provide anonymity. Here the term 'email address' is used for simplicity.

3.2 The Mailrank Server

To filter unwanted messages, a framework is needed that does all the work. SpamAssassin is predestined for that.

In this section I give an overview of SpamAssassin and the plugin for it. Then I introduce the Mailrank server software and describe the details of the implementation.

3.2.1 SpamAssassin

SpamAssassin [16] is a powerful filter program for unwanted messages (Spam). It uses header and text analysis, Bayesian filtering, DNS blocklists[2] and collaborative filtering databases[3] for the identification of Spam.

SpamAssassin is written in perl and extensible with plugins. Figure 3.1 gives an overview of the architecture.

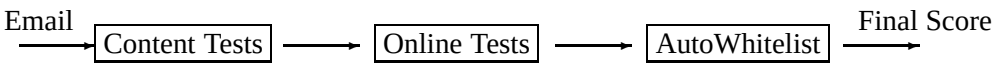


Figure 3.1: The SpamAssassin Architecture

3.2.1.1 The Tests

If an email is passed through the SpamAssassin engine, it is scanned with various tests. A tests can be used in four different ways:

1. **Local:** No DNS blocklist or other online tests are made. This is useful when no Internet connection is available.
2. **Net:** In addition to the content analysis, DNS blocklist servers are asked and the hash of the email is sent to a collaborative filtering database like Razor [17] or DCC [18]

[2] A mail server at the internet, that is known to send Spam, is listed in these blocklists and other mailservers can get this data to reject connections from these mail servers. In SpamAssassin this is used to get an additional criteria for judging the mail as Spam or not.

[3] If a user identifies an email as Spam, he can send a hashed version of this email to a server running the collaborative filtering database. Then this information can be used by a plugin in SpamAssassin to classify an email as Spam. Familiar examples are Razor and DCC.

3. **With Bayes:** The basic idea behind Bayesian filtering is that a Spam email is divided into tokens, these are words or short phrases. The tokens are stored and can be reused to identify similar tokens in another email. As time grows the filter "learns" more and more tokens and a email that includes several of these bad words can be sure classified as Spam.

4. **With Bayes and Net:** This is a combination of 2 and 3 and is the test method with the highest detection rate of Spam.

3.2.1.2 The AutoWhitelist

The AutoWhitelist (AWL) is a score averaging system. It keeps track of the historical average of a sender and pushes any subsequent mail towards that average. It works with an algorithm that is started after every other test is made.

Let *SCORE* be the score for an email after the tests. Every sender has an entry in the AutoWhitelist, together with the long-term score *MEAN*. This score is the sum of all scores made in the past for this sender (*TOTAL*) divided by the count of all received mails from this sender (*COUNT*).

The portion of the AutoWhitelist on the final score for this email (*FINALSCORE*) can be changed with the AutoWhitelist factor (*FACTOR*).

$$MEAN = \frac{TOTAL}{COUNT}$$

$$FINALSCORE = SCORE + (MEAN - SCORE) * FACTOR$$

3.2.1.3 Plugin for Spamassassin

One of the goals of Mailrank is to provide a plugin for SpamAssassin that automatically checks an email, asks the Mailrank server for a trust value and incorporate this into the tests. The needed data for this computation are in the AutoWhitelist.

- The email address from the sender of an email
- The IP of the server where the email comes from
- The score of that sender, computed as described above
- The number of emails received from this sender

3.2.2 Mailrank Server Overview

For ranking email addresses, a piece of software is needed that does all the work. It has to take the addresses and belonging data, has to make the computation and then send the trust value back to the sender.

Data can be sent to and received from the server in two possible ways: either via a socket connection or via email. The corresponding modules are a socket server (MRSocket) and an email server (MRMail).

A MySQL database is used as a storage backend (MRMySQLDatabase) and is accessible via the database interface (MRDatabaseHandler).

Figure 3.2 shows an overview of the Mailrank server software.

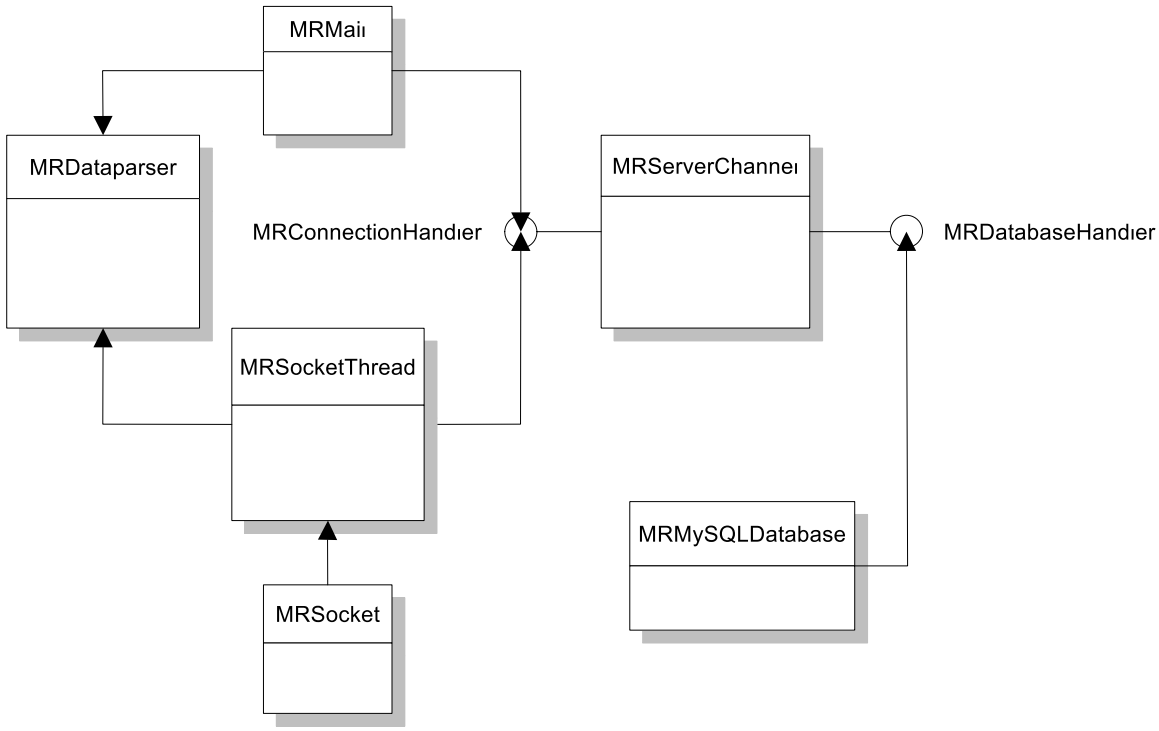


Figure 3.2: An Overview of the Mailrank Server

3.2.3 The Data Representation: MRData

As described in 3.2.1.3 the data required for computation are provided by the Mailrank plugin for SpamAssassin. Additionally the email address of the user sending this information is added as well as a command, that says what should happen with the data. Figure 3.3 shows an example, where a user (`goodguy@good.com`) sends data about a spammer (`spam@spammer.com`) with the score (`30.43`) and count of emails received from this spammer (`10`) to the Mailrank server, which should store it (`setValues`).

```
setValues:goodguy@good.com:spam@spammer.com:30.43:10
```

Figure 3.3: Example of data sent to the Mailrank server

The Mailrank plugin provides anonymity, therefore every email address is replaced by it's hex representation (figure 3.4).

```
setValues:fda86d315ff7c78db67e8:3fa05b297c4dfa4d4fe3c:30.43:10
```

Figure 3.4: Example of anonymized data

For the JAVA implementation these data are encapsulated in one class called MRData. A short clipping with the most important methods of the class is shown in figure 3.5 (The complete API can be found in appendix A).

```
public class MRData {
    /**
     * A command that can be one of the following:
     * getValues, getEntryByUser, getEntryBySpamaddress,
     * getEntryByScore, getEntryByCount
     * The getEntryBy* commands are just for testing purposes
     */
    private String command;

    private double score;
    /**
     * A count is always an integer and reflects the
     * count of mail I’ve received from
     * a specific sender.
     */
    private int count;
    /**
     * The hash of the emailaddress of user sending this data
     */
    private String user;
    /**
     * The hash of the spam-mailaddress of a sender,
     * sending me unwanted messages
     */
    private String address;

    MRData(String command, String user, String address,
        double score, int count) {
        this.command = command;
        this.user = user;
        this.address = address;
        this.score = score;
        this.count = count;
    }

    MRData(String user, String address, double score, int count) {
        this.user = user;
        this.address = address;
        this.score = score;
        this.count = count;
    }
```

Figure 3.5: The MRData class

3.2.4 The MRDataParser

The MRDataParser class provides methods for parsing a given string and returns an MRData object, if the string is recognized as valid. Figure 3.6 shows the structure of a string expected by the parser.

Not only the structure of the string, but also the data type of the fields is verified (see table 3.1).

```
Command:HashSenderEmailAddress:HashSpammerEmailAddress:Score:Count
```

Figure 3.6: Structure of a valid String

Field	Datatype	Notes
Command	String	only: setValues, getEntryByUser, getEntryBySpamaddress, getEntryByScore, getEntryByCount
HashSenderEmailAddress	String	
HashSpammerEmailAddress	String	
Score	Double	
Count	Integer	

Table 3.1: Datatypes expected by the MRDataParser

3.2.5 Abstraction

Interfaces are used to keep an implementation flexible. The Mailrank server includes two interfaces: **MRConnectionHandler** and **MRDatabaseHandler**.

For the communication between these two interfaces an adaptor class (MRServerChannel) is used, that translates the calls from one to the other interface.

3.2.5.1 MRConnectionHandler

This interface is used to abstract from the communication channel. Only one method is needed to implement this interface:

public Vector send(MRData myData);

The information always flows from the servers (MRSocket/MRMail) to the database. Only the answer to a query is sent back.

3.2.5.2 MRDatabaseHandler

This interface is used to abstract from the type of a database. Currently only a MySQL database engine is implemented, but it's also possible to use for instance an Oracle database. Table 3.2 shows the methods that have to be implemented.

Name of Method	Description
public boolean init()	initializes the database connection
public boolean close()	closes the database connection
public void setValues(MRData myData)	sends a MRData object to the database
public Vector getEntryByUser(MRData myData)	sends a MRData object to the database including a hash of a user email address and returns all entries from that user in the database
public Vector getEntryBySpamaddress(MRData myData)	sends a MRData object to the database including a hash of a Spam email address and returns all entries from that Spam address in the database
public Vector getEntryByScore(MRData myData)	sends a MRData object to the database including a score and returns all entries from that score in the database
public Vector getEntryByCount(MRData myData)	sends a MRData object to the database including a count number and returns all entries from that count in the database

Table 3.2: Method Definitions of MRDatabaseHandler

3.2.5.3 MRServerChannel

The two interfaces MRDatabaseHandler and MRConnectionHandler need to talk to each other. The method calls have to be "translated". Such a class is called **adaptor class**.

The MRServerChannel provides a method (`doWork()`), that takes an MRData object, handles all the database issues and returns an a vector of MRData objects.

3.2.6 The Socket Server: MRSocket

MRSocket is a threaded server. That means every time a connection attempt is made, the server starts a new thread (MRSocketThread) that does

all the work. Figure 3.7 and 3.8 show flow diagrams, explaining how the server works.

When a new MRSocketThread is started, it receives the data from the socket and creates a new instance of MRDataParser, which parses the given string and returns a MRData object. This is sent through the MRServerChannel. The result is a vector of MRData objects, which are converted into strings and sent to the client. After that the thread ended.

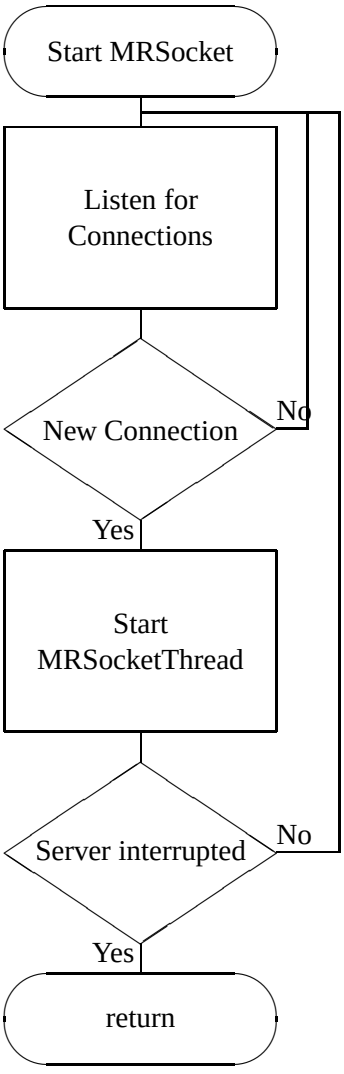


Figure 3.7: Data Flow in MRSocket

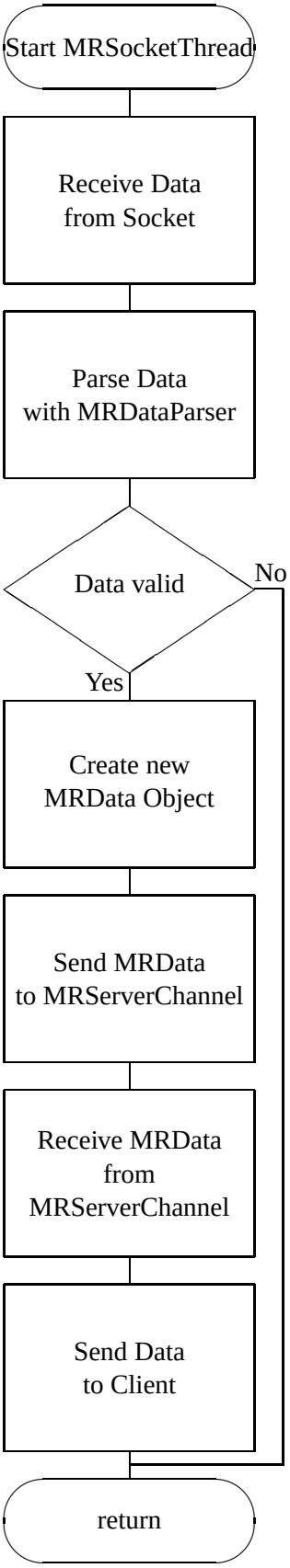


Figure 3.8: Data Flow in MRSocketThread

3.2.7 The Mail Server: MRMail

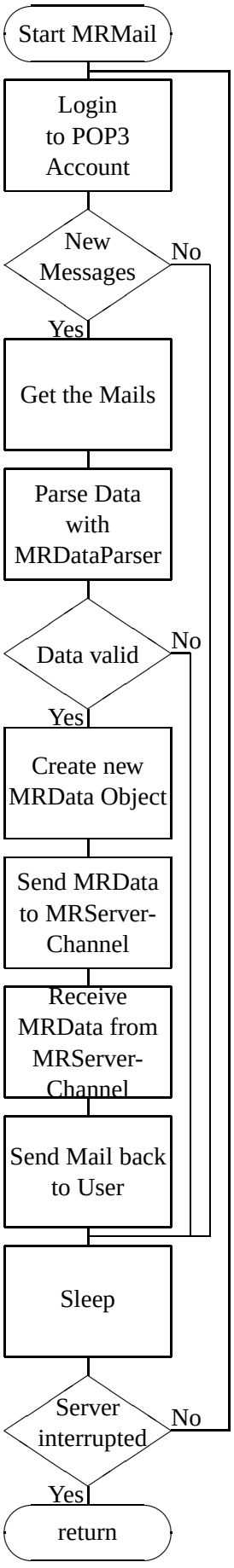
The Mailrank server is capable of handling different types of connections to clients. In addition to socket connections, emails can be used for sending data to and receiving data from the Mailrank server.

The MRMail class handles all the email communication. It provides methods for sending and getting emails. The server uses the POP3 [19] for getting emails. So, it is not an email server the traditional way, but there is better and approved software like Postfix [20] and Cyrus [21] to handle emails very effective and I decided not to reinvent the wheel.

Figure 3.9 on site 34 shows an overview of the email communication in MRMail and interaction with other classes.

When the server is started it logs on to the pop3 account and receives the email. If there are no new messages, it goes to sleep a certain time. Otherwise the server gets the emails from the pop3 account and creates a new instance of MRDataParser. If the data are valid, a new MRData object is created by the parser and send by the server through the MRServerChannel, who returns a vector of MRData objects. From this vector the MRMail creates a new email and send it back to the user.

After this all is done, the server goes to sleep again, if not interrupted.



Data Flow MRMail

3.2.8 The Database Engine: MRDatabase

For storing data in the Mailrank server MySQL ([22]) is used. MySQL is a multithreaded, multi-user, SQL (Structured Query Language) relational database server. It is widely used and proven for scalability and reliability.

The methods of the database (see table 3.2) are called through the database interface MRDatabaseHandler. They are translated into SQL statements and send to the database, which returns a result set. For every result of this result set a MRData object is created and added to a result vector. Finally this result vector is sent through the MRDatabaseHandler interface.

Figure 3.10 shows an example of a method call translated into a SQL statement.

setvalues(MRData mydata):

```
myResultSet = "SELECT users.ID, Count, Score" +
                    " FROM users,spam WHERE" +
                    " User = \"" + mydata.getUser() + "\"" +
                    " AND " +
                    " users.ID=spam.ID;"

START TRANSACTION;
UPDATE spam SET" +
                    " Score = " + mydata.getScore() + "," +
                    " Count = " + mydata.getCount() +
                    " WHERE " +
                    " ID=" + myResultSet.getString("ID") + " AND" +
                    " Address = " + "\"" + mydata.getAddress() + "\"" +
                    " ;");
COMMIT;
```

Figure 3.10: Method call translated into a SQL statement

Chapter 4

Further Work

The thesis gave an introduction of the Mailrank approach. The algorithm will be developed in detail and a paper will be written, that discuss all the facets of Mailrank. Also a comparison of the algorithms in chapter 2 to Mailrank will be made.

Furthermore the Mailrank server will be extended to it's full functionality and intensive test will be made.

Hopefully this all helps us to get rid of Spam and make our communication more effective and less bothering.

Appendix A

Javadoc

A.1 Mailrank Hierarchical Index

A.1.1 Mailrank Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

A.2 Mailrank Class Index

A.2.1 Mailrank Class List

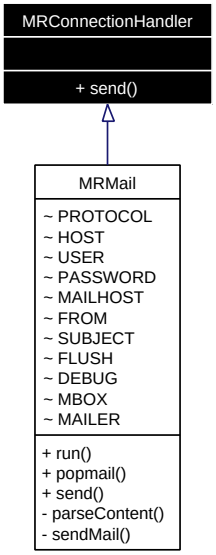
Here are the classes, structs, unions and interfaces with brief descriptions:

A.3 Mailrank Class Documentation

A.3.1 MRConnectionHandler Interface Reference

Inherited by **MRMail**, and MRSocketThread.

Inheritance diagram for MRConnectionHandler:



Public Member Functions

- Vector **send** (**MRData** myData)

A.3.1.1 Detailed Description

Abstracts from the connection. Socket and mail implemented now. See **MRSocket**(p. 52) and **MRMail**(p. 44)

A.3.1.2 Member Function Documentation

Vector MRConnectionHandler.send (MRData *myData*)

sends the Data into the Server and receives if appropriate the query result. The vector consists of **MRData**(p. 39) Objects

Implemented in **MRMail** (p. 46).

The documentation for this interface was generated from the following file:

- src/MRConnectionHandler.java

A.3.2 MRData Class Reference

Package Functions

- **MRData** (String command, String user, String address, double score, int count)
- **MRData** (String user, String address, double score, int count)

A.3.2.1 Detailed Description

This class encapsulate all the data. Must be extended when trustvalues implemented.

A.3.2.2 Constructor & Destructor Documentation

MRData.MRData (String *command*, String *user*, String *address*, double *score*, int *count*) `[package]`

This constructor is used, when querying the DB. All the data can be get out of the autowhitelist of spamassassin see the perlclients included in this distribution

Parameters:

command getValues, getEntryByUser, getEntryBySpamaddress, getEntryByScore, getEntryByCount

user the hash of the usermailaddress

address the hash of the spam-mailaddress

score the score (see autowhitelist in spamassassin)

count the count (see autowhitelist in spamassassin)

MRData.MRData (String *user*, String *address*, double *score*, int *count*) `[package]`

This constructor is used, when receiving the result of a query. All the data can be get out of the autowhitelist of spamassassin see the perlclients included in this distribution

Parameters:

user the hash of the usermailaddress

address the hash of the spam-mailaddress

score the score (see autowhitelist in spamassassin)

count the count (see autowhitelist in spamassassin)

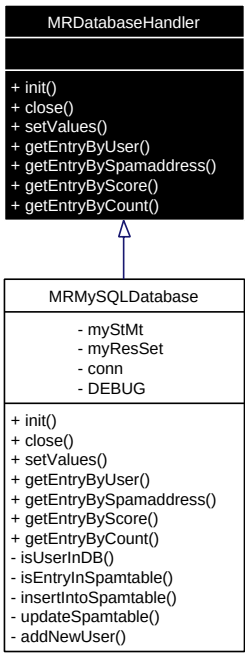
The documentation for this class was generated from the following file:

- src/MRData.java

A.3.3 MRDatabaseHandler Interface Reference

Inherited by **MRMySQLDatabase**.

Inheritance diagram for MRDatabaseHandler:



Public Member Functions

- boolean **init** ()
- boolean **close** ()
- void **setValues** (**MRData** myData)
- Vector **getEntryByUser** (**MRData** myData)
- Vector **getEntryBySpamaddress** (**MRData** myData)
- Vector **getEntryByScore** (**MRData** myData)
- Vector **getEntryByCount** (**MRData** myData)

A.3.3.1 Detailed Description

Created by IntelliJ IDEA. User: broso Date: Aug 10, 2004 Time: 7:35:54 PM To change this template use File | Settings | File Templates.

A.3.3.2 Member Function Documentation

boolean MRDatabaseHandler.close ()

Closes the DB Connection

Returns:

true if all goes well, otherwise throws an SQLException

Implemented in **MRMySQLDatabase** (p. 48).

Vector MRDatabaseHandler.getEntryByCount (MRData *myData*)

Testmethod, gets all entries of a specific count

Parameters:

myData a **MRData**(p. 39) object

Returns:

a vector of **MRData**(p. 39) objects

Implemented in **MRMySQLDatabase** (p. 49).

Vector MRDatabaseHandler.getEntryByScore (MRData *myData*)

Testmethod, gets all entries of a specific score

Parameters:

myData a **MRData**(p. 39) object

Returns:

a vector of **MRData**(p. 39) objects

Implemented in **MRMySQLDatabase** (p. 49).

Vector MRDatabaseHandler.getEntryBySpamaddress (MRData *myData*)

Testmethod, gets all entries of a specific spam mailaddress

Parameters:

myData a **MRData**(p. 39) object

Returns:

a vector of **MRData**(p. 39) objects

Implemented in **MRMySQLDatabase** (p. 49).

Vector MRDatabaseHandler.getEntryByUser (MRData *myData*)

Testmethod, gets all entries sent by a specific user

Parameters:

myData a **MRData**(p. 39) object

Returns:

a vector of **MRData**(p. 39) objects

Implemented in **MRMySQLDatabase** (p. 49).

boolean MRDatabaseHandler.init ()

Init the connection to the database. Please edit the parameters to fit our needs.

Returns:

true if the connection works, otherwise throws an exception

Implemented in **MRMySQLDatabase** (p. 50).

void MRDatabaseHandler.setValues (MRData *myData*)

Writes the data (the data inside a **MRData**(p. 39) object) into the database

Parameters:

myData a **MRData**(p. 39) object

Implemented in **MRMySQLDatabase** (p. 50).

The documentation for this interface was generated from the following file:

- src/MRDatabaseHandler.java

A.3.4 MRDataParser Class Reference

Public Member Functions

- **MRData parse** (String line)

Static Package Attributes

- final boolean **DEBUG** = false

A.3.4.1 Detailed Description

Parses given strings, that look like: command:userhash:spamhash:score:count or command:{userhash|spamhash|score|count}

A.3.4.2 Member Function Documentation

MRData MRDataParser.parse (String *line*)

Parses a line that have to look like: in the case of setValues: setValues:userhash:spamhash:score:count in the case of getEntryByUser: getEntryByUser:userhash in the case of getEntryBySpamaddress: getEntryBySpamaddress:spamhash in the case of getEntryByScore: getEntryByScore:score in the case of getEntryByCount: getEntryByCount:count Otherwise the String will be discared and an empty **MRData**(p. 39) object is returned For a new command this parse have to be extended

Parameters:

line a String that fullfills the above requirements

Returns:

a **MRData**(p. 39) object with the data given, in the case of malformed data empty

A.3.4.3 Member Data Documentation

final boolean MRDataParser.DEBUG = false `[static, package]`

if true, more output - useful for debugging

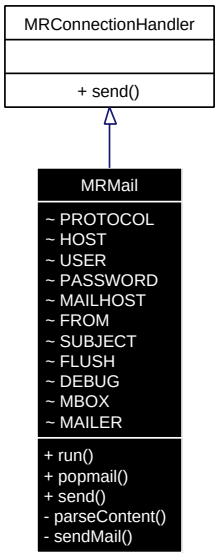
The documentation for this class was generated from the following file:

- src/MRDataParser.java

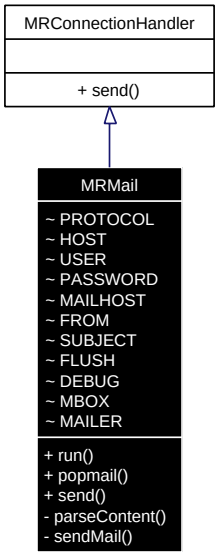
A.3.5 MRMail Class Reference

Inherits **MRConnectionHandler**.

Inheritance diagram for MRMail:



Collaboration diagram for MRMail:



Public Member Functions

- void **run** (int minutes)
- void **popmail** ()
- Vector **send** (**MRData** myData)

Package Attributes

- final String **PROTOCOL** = "pop3s"
- final String **HOST** = "mypop3host"
- final String **USER** = "mypop3username"
- final String **PASSWORD** = "mypop3password"
- final String **MAILHOST** = "mysmtpserver"
- final String **FROM** = "mymailaddress"
- final String **SUBJECT** = "Your Mailrank results"
- final boolean **FLUSH** = false
- final boolean **DEBUG** = false

A.3.5.1 Detailed Description

All the functionality for handeling mails is here This server needs a POP3 account somewhere and a SMTP server for sending mails Please look at INSTALL

A.3.5.2 Member Function Documentation

void MRMail.popmail ()

Does all the work gets a connection to the POP3 server, receives the mails, processes them and sends out answer mails

void MRMail.run (int *minutes*)

Runs **popmail()**(p. 45) and sleeps for minutes given

Parameters:

minutes the interval between two "pops"

Vector MRMail.send (MRData *myData*)

Takes a **MRData**(p. 39) object and asks the DB. Gets back a vector of **MRData**(p. 39) objects

Parameters:

myData a **MRData**(p. 39) object

Returns:

a Vector of **MRData**(p. 39) objects

Implements **MRConnectionHandler** (p. 39).

A.3.5.3 Member Data Documentation

final boolean MRMail.DEBUG = false `[package]`

if true, more output - useful for debugging

final boolean MRMail.FLUSH = false `[package]`

flushes the mails after processing them Set this true (recommended) or the mails will be processed again and again

final String MRMail.FROM = "mymailaddress" `[package]`

Address for sending emails (FROM:)

final String MRMail.HOST = "mypop3host" `[package]`

The POP3 host, where the account is

final String MRMail.MAILHOST = "mysmtpserver" `[package]`

The smtp server for sending mails

final String MRMail.PASSWORD = "mypop3password" `[package]`

The Password

final String MRMail.PROTOCOL = "pop3s" `[package]`

The Postoffice mailprotocol. Please use the secure variant: pop3s

final String MRMail.SUBJECT = "Your Mailrank results" `[package]`

Subject of the mails going out

final String MRMail.USER = "mypop3username" `[package]`

The username

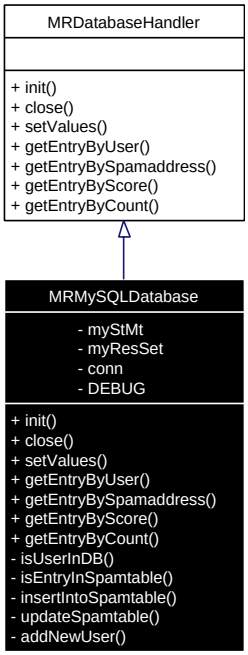
The documentation for this class was generated from the following file:

- src/MRMail.java

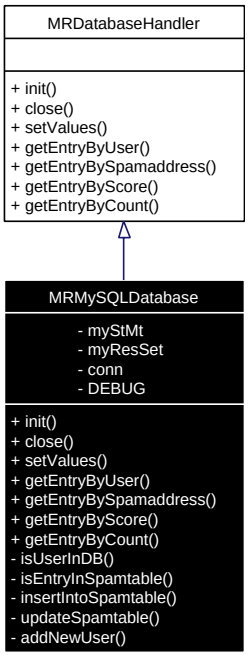
A.3.6 MRMySQLDatabase Class Reference

Inherits **MRDatabaseHandler**.

Inheritance diagram for MRMySQLDatabase:



Collaboration diagram for MRMySQLDatabase:



Public Member Functions

- boolean **init** ()
- boolean **close** ()
- void **setValues** (**MRData** myData)
- Vector **getEntryByUser** (**MRData** myData)
- Vector **getEntryBySpamaddress** (**MRData** myData)
- Vector **getEntryByScore** (**MRData** myData)
- Vector **getEntryByCount** (**MRData** myData)

A.3.6.1 Detailed Description

This class implements the connection to a MySQL Server

A.3.6.2 Member Function Documentation

boolean MRMySQLDatabase.close ()

Closes the DB Connection

Returns:

true if all goes well, otherwise throws an SQLException

Implements **MRDatabaseHandler** (p. 41).

Vector MRMySQLDatabase.getEntryByCount (MRData *myData*)

Testmethod, gets all entries of a specific count

Parameters:

myData a **MRData**(p. 39) object

Returns:

a vector of **MRData**(p. 39) objects

Implements **MRDatabaseHandler** (p. 41).

Vector MRMySQLDatabase.getEntryByScore (MRData *myData*)

Testmethod, gets all entries of a specific score

Parameters:

myData a **MRData**(p. 39) object

Returns:

a vector of **MRData**(p. 39) objects

Implements **MRDatabaseHandler** (p. 41).

Vector MRMySQLDatabase.getEntryBySpamaddress (MRData *myData*)

Testmethod, gets all entries of a specific spam mailaddress

Parameters:

myData a **MRData**(p. 39) object

Returns:

a vector of **MRData**(p. 39) objects

Implements **MRDatabaseHandler** (p. 42).

Vector MRMySQLDatabase.getEntryByUser (MRData *myData*)

Testmethod, gets all entries sent by a specific user

Parameters:

myData a **MRData**(p. 39) object

Returns:

a vector of **MRData**(p. 39) objects

Implements **MRDatabaseHandler** (p. 42).

boolean MRMySQLDatabase.init ()

Init the connection to the database. Please edit the parameters to fit our needs.

Returns:

true if the connection works, otherwise throws an exception

Implements **MRDatabaseHandler** (p. 42).

void MRMySQLDatabase.setValues (MRData *myData*)

Writes the data (the data inside a **MRData**(p. 39) object) into the database

Parameters:

myData a **MRData**(p. 39) object

Implements **MRDatabaseHandler** (p. 42).

The documentation for this class was generated from the following file:

- src/MRMySQLDatabase.java

A.3.7 MRServer Class Reference

Static Public Member Functions

- void **main** (String[] args)

A.3.7.1 Detailed Description

The mainclass for the whole project

A.3.7.2 Member Function Documentation

void MRServer.main (String[ ] *args*) `[static]`

starts a new instance of **MRSocket**(p. 52) and **MRMail**(p. 44) todo: using 2 Threads, so the 2 parts can be started in parallel

Parameters:

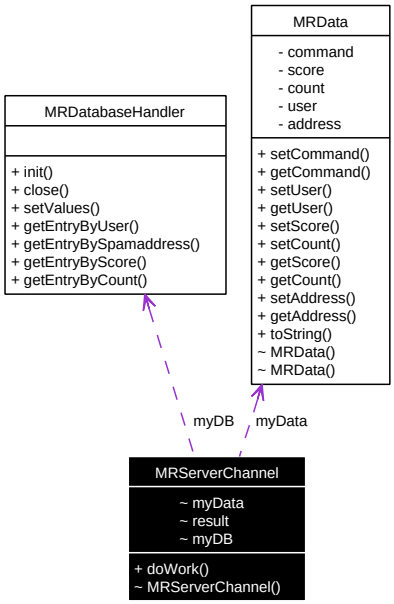
args not used

The documentation for this class was generated from the following file:

- src/MRServer.java

A.3.8 MRServerChannel Class Reference

Collaboration diagram for MRServerChannel:



Public Member Functions

- Vector **doWork** ()

A.3.8.1 Detailed Description

This is the adaptorclass to abtract from the communication channel It handles all the communication between the **MRDatabaseHandler**(p. 40) interface and the **MRConnectionHandler**(p. 38) interface

A.3.8.2 Member Function Documentation

Vector MRServerChannel.doWork ()

Like the name this method does all the work. In case of extending the **MRData**(p. 39) class (trustvalue), this method must be extended

Returns:

a Vector that consists of **MRData**(p. 39) objects

The documentation for this class was generated from the following file:

- src/MRServerChannel.java

A.3.9 MRSocket Class Reference

Public Member Functions

- void **init** (int port)
- boolean **closeConnection** (ServerSocket socket)

A.3.9.1 Detailed Description

Class for the Socketserver

A.3.9.2 Member Function Documentation

boolean MRSocket.closeConnection (ServerSocket *socket*)

close the socket

Parameters:

socket the Serversocket

Returns:

true if all goes well

void MRSocket.init (int *port*)

Binds a port to a socket and wait for connections if a connection attempt happend it starts a thread, that handles all

Parameters:

port the listen port

The documentation for this class was generated from the following file:

- src/MRSocket.java

Appendix B

Installation and Usage

B.1 Installation

Please download the Mailrank software from the peertrust cvs (cvs.sourceforge.net/cvsroot/peertrust) or use the cvs snapshot included with this thesis.

Please read the following instructions carefully.

1. You need several jar-files:
   a) get the latest mysql-database connector for java from mysql.com (`http://www.mysql.com/products/connector/j/`)
   b) get the latest javamail classes (`http://java.sun.com/products/javamail/`) Mailrankserver is known to work with JavaMail 1.3.2 Early Access release
   c) get the latest JavaBeans activation framework, because JavaMail needs it (`http://java.sun.com/products/javabeans/jaf/`)

   If you like put these files in your classpath.
2. Get the source from cvs (probably you already have them now)
3. Get an pop3 account for the server
4. Insert the account data into MRMail (later with properties) Please use pop3s and don't send your password arround in the clear. It's worth it.
5. Get the ca certificate if the pop3 server uses a self-signed certificate. Add this certificate to your keystore with:

```
# keytool -import -file /path/to/cacert.pem
-trustcacerts -storetype jks
```

password is usually: changeit
List the keystore with:

```
# keytool -list -trustcacerts -storetype jks
```

and make sure that it's listed

6. Edit MRMail again and insert the path to the keystore (usually `YOURHOME/.keystore`) While you there edit the flush boolean variable: true: all the mails that are processed will be deleted from the server false: the mail be not be deleted, useful for debugging only

7. Insert the remaining parameter (like smtphost etc.) into MRMail

8. You need a MySQL account, please enter the parameter into MRMySQLDatabase (line 37)

9. Use the file mysql.sql in supportfiles to create the necessary tablestructure (take a look at it before!)

10. Change the path to the properties file in MRServer. The Properties file is in supportfiles While you there, set the port, 4444 should work in most cases

11. We're almost done: Take a look at `start.sh` in supportfiles and make sure, that the paths fit your environment. Then the big moment: compile the server and start it with this script.

B.2 Usage

I assume you have the server up and running.

First, try to receive a pop3 mail. You can do this by using your mailprogramm to send a mail to the mailaddress (the pop3 mailbox) of the server. Please use only plaintext mails. The subject doesn't matter, but the body have to look like this:

```
setValues:my@email.com:spam@spammer.org:-2.34:43
setEntryByUser:my@email.com
```

You should then receive a mail with the subject "Your Mailrank results" and the values in it. If not, turn on debugging by setting the debug boolean value to true in the MRMail and look at the output.

Try to use the MRSocket server. In the perl directory you find among other things `getEntry.pl` and `setEntry.pl`. At the beginning of this two files, you need to set the host and port to connect to:

```
my $remote_host="123.456.789.087";
my $remote_port=4444;
```

In most cases the port 4444 should work. In the Terminal try:

```
./setEntry my@mail.com spam@spammer.org 123.456 345
```

if this works, try to receive the entry you just added:

```
getEntry my@mail.com
```

You should see an output.

Thats it for the moment.

If you ran into problems and cannot help yourself, please write an email to: brosowski@l3s.de I will help you.

Bibliography

[1] R. Guha, R. Kumar: Propagation of Trust and Distrust, In *Proceedings of the 13th International WWW Conference (WWW 2004)*, 2004

[2] J. Golbeck, J. Hendler: Inferring Reputation on the Semantic Web, In *Proceedings of the 13th International WWW Conference (WWW 2004)*, 2004

[3] S. Kamvar, M.Schlosser and H. Garcia-Molina: The EigenTrust Algorithm for Reputation Management in P2P Networks, In *Proceedings of the 12th International WWW Conference 2003*, 2003

[4] L.Page, S. Brin, R. Motwani and T. Winograd: The PageRank Citation Ranking: Bringing Order to the Web. Technical Report, Stanford University, 1998

[5] C. Ziegler, G. Lausen: Spreading activation models for trust propagation. In *Proceedings if the IEEE International Conference on e-Technology, e-Commerce and e-Service*, 2004

[6] Lik Mui, Mojdeh Mohtashemi, Ari Halberstadt: A Computational Model of Trust and Reputation, In *Proceedings of the 35th Hawaii International Conference on System Sciences*

[7] E. Ostrom, A Behavioral Approach to the Rational-Choice Theory of Collective Action, In *American Political Science Review, 91(1), pp. 1-22*

[8] Bruce Schneier: *Applied cryptography : protocols, algorithms, and source code in C*, New York, Wiley, 1994

[9] International Telecommunication Union - Standardization Sector (ITU-T), The X.509 Standard, *http://www.itu.int/rec/recommendation.asp?type=folders&parent=T-REC-X.509*

[10] PeerTrust: Automated Trust Negotiation for Peers on the Semantic Web, *http://www.l3s.de/peertrust*

[11] EDUTELLA: RDF-based Metadata Infrastructure for P2P Applications, *http://edutella.jxta.org*

[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker: A scalable content-addressable network, In *Proceedings of ACM SIGCOMM*, August 2001

[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrsihnan: Chord: A scalable peer-to-peer lookup service for internet applications, In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications*, pages 149-160, ACM Press, 2001

[14] PGP.com, *An Introduction to Cryptography*, October 2003

[15] Mike Ashley, *The GNU Privacy Handbook*, 1999, The Free Software Foundation

[16] The SpamAssassin Project Page: *http://spamassassin.apache.org*

[17] Vilpul's Razor a distributed, collaborative, spam detection and filtering network: *http://razor.sourceforge.net*

[18] DCC: Distributed Checksum Clearinghouse, *http://www.rhyolite.com/anti-spam/dcc*

[19] Post Office Protocol - Version 3, *http://www.ietf.org/rfc/rfc1939.txt*

[20] Postfix - an alternative to the widely-used Sendmail program, *http://www.postfix.org*

[21] The Cyrus IMAP Server, *http://asg.web.cmu.edu/cyrus/imapd/*

[22] MySQL Database Server, *http://www.mysql.com*

[23] E. Smith, S. Nolen-Hoeksema, D. Frederickson and G. Loftus: *Atkinson and Hilgard's Introduction to Psychology*, Thomason Learning, Boston, Ma, USA, 2003