ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Spamato Reloaded

Trust, Authentication and More in a Collaborative Spam Filter System

Simon Schlachter

Master's Thesis

February 18, 2004 – August 17, 2004

Supervising Professor: Prof. Dr. Roger Wattenhofer
Supervising Assistant: Keno Albrecht

Preface

Abstract

SPAMATO is a collaborative spam filter system implemented in Java. It is designed as a framework to support any number and kind of spam filters. The initial version features an *URL Filter*, which extracts URLs from incoming mail messages and calculates a fingerprint based on these URLs. This fingerprint is compared to a central database. If its fingerprint is known as spam, the mail message is classified as spam. Although SPAMATO is written in Java, a Microsoft Outlook Add-In has been written in Visual Basic.NET to use SPAMATO in everyday's work.

This thesis takes SPAMATO several steps further. First, a lot of improvements and extensions are introduced to the existing system, such as a new user interface in Outlook, a new user interface to configure filters, a new engine providing interesting statistics about the incoming mail of the user, multi-user capabilities as well as more consistent and robust communication engines to create XML messages.

Second, a new spam filter is implemented and added to SPAMATO—the *Razor/SpamNet Filter*. Razor itself is also a collaborative spam filter network. It picks random text passages from mail messages and calculates a fingerprint using them. This fingerprint is used to report spam messages to the Razor network and to check if a message is known as spam. Using the Razor/SpamNet Filter, SPAMATO is able to check incoming mail using the Razor network in addition to the URL Filter network.

Third, the *Automatic Authentication and Authorization System (AAAS)* is presented. AAAS uses email adresses and key sets instead of username and password to authenticate users. Therefore, no user action is needed in order to register a new account or to reregister an existing account after reinstalling. It is even possible to use the same account on any number of systems at the same time.

Fourth, a new trust system is introduced. Trust systems are designed to protect collaborative systems against malicious users. Most of today's trust systems, however, use a kind of AIMD (additive increase, multiplicative decrease) approach and therefore fail if no global classification of a mail message can be achieved. This thesis presents TROOTH, an advanced trust system which is able to handle even these messages correctly by using a new approach for cases where no global decision is possible.

Acknowledgment

Completing my degree by writing this thesis, I would like to thank my family and all my friends for their support and help during all the years of my study. A lot of my time has been consumed by doing exercises, learning for examinations and, during the last months, writing this thesis. Therefore, I apologize to everyone who had to do without me due to my lack of time.

I would like to thank Nicolas Burri for designing SPAMATO—offering me the opportunity to write this thesis. He is also the source of a lot of ideas for the systems we designed during this thesis.

I am very grateful to Keno Albrecht, my supervisor, for supporting me during the last six months. We had a lot of fun and good ideas while discussing, having a break, and in our weekly "SPAMATO sessions". I was very happy to receive your "how is it going"-phone call while you were on holiday.

Thank you, Prof. Wattenhofer, for putting me in the position to write my thesis under your supervision and for your ideas concerning SPAMATO and TROOTH.

Last but not least, I am indepted to Aaron Zollinger and Claudia Gamma for proof-reading my thesis—and to everyone who motivated and supported me without being mentioned by name, you all know who you are. Thank you.

Contents

Sshh, dear, don't cause a fuss. I'll have your spam. I love it. I'm having spam spam spam spam spam spam spam baked beans spam spam spam and spam!

Monthy Python *(The Spam Skit)*

1 Introduction

> Once upon a time there was a young girl, who was so much beloved by her father that she received a computer for her birthday, and, some days later, it was connected to a worldwide network called "internet". As the girl, in a transport of delight, soon found out, this network provided the possibility to write messages to her friends without the need for pen and paper, "email" this was called.
>
> Now it so happened that one day, when the young girl gleefully checked her mail, longing for her boyfriend's answer to her last message, she did not receive only one message but hundreds. The girl jumped for joy, for she had never received more than one message a day before. But, as she soon found out, these were hundreds of messages from senders *she* did not know but who pretended to know *her*. She was offered to buy medicals, software or even college degrees. Since the girl was not interested in spending her money on anything beside clothes, which none of these messages offered, she decided to delete them. This was the reason why she never received her boyfriend's answer—for it was amongst the messages the girl deleted.

Although this fairytale is invented, it intimates several problems of today's email communication. Some years ago, no one having an email address ever received unsolicited mail messages (known as *spam*) except for newsletters he or she did not want. Today, according to statistics [UNs04], 75% of all email users spend more than ten minutes dealing with spam—daily. More than 80% of the children, who are using email, receive spam on a daily basis. Furthermore, the number of spam messages which are sent over the internet has doubled every six months in the last two years. Though impressive these numbers may be, they could even be worse: 70% of spam messages get filtered by state-of-the-art spam filters and are therefore not delivered to mail boxes at all. If it were not for spam filters, email would no longer be useful to anyone but people who send spam (referred to as *spammers*).

The downside of spam filters however, are too many legitimate mail messages that are falsely identified as spam (these messages are reffered to as *false positives*). According to statistics, one in eight messages identified as spam is in fact a false positive. Therefore, the process of filtering unsolicited mail can not be fully automated. Someone has to supervise the spam filters and check the messages they filter out in order not to lose too many falsely classified mail messages. This problem has been pointed out by Juvenal a long time ago:

> Sed quis custodiet ipsos Custodes?

(But who will guard the guardians?)

—Juvenal, 60 to 130 AD

Unlike other spam filter approaches, collaborative spam filters, such as SPAMATO, accept the fact that no automatic spam filter will ever be able to correctly identify every sort of spam mail. Collaborative filters, therefore, do not classify messages based on some sort of rule (or auto-learning algorithm) but let users classify messages themself. They connect users to a community in which human beings classify messages very much like in a voting known from politics.

Like in politics, the identification of voters must be assured to achieve a proper voting result. In collaborative systems, even more assertions need to be guaranteed. The system needs to take care of how users register, how they are identified in order to be able to vote, and how malicious users, who are trying to influence the voting result to fit their needs, are detected and prevented from harming the votings.

This thesis solves the problems mentioned above and introduces other extensions and improvements to the existing SPAMATO system. Chapter 2 provides an overview of all these extensions. Then, a new spam filter, which connects to the network of the Razor/SpamNet community to identify spam, is described in Chapter 3. Chapter 4 presents *AAAS*, the *Automatic Authorization and Authentication System*. AAAS registers users to a collaborative system without the need of any interaction by the user. These accounts are portable, providing the possibility to use the same account on more than one machine—again, no user interaction is required to achieve this. Only registered users are allowed to take an active part in votings regarding the classification of mail messages. In Chapter 5, TROOTH, an advanced trust system, is introduces. TROOTH is superior to currently used trust systems, based on the *AIMD* (additive increase, multiplicative decrease) approach, because it is able to handle votings, which do not achieve a clear majority, in a more sophisticated way. Finally, a summary and an outlook for subsequent theses are provided.

2 Spamato Revolutions: Extensions and Improvements

This thesis is based on SPAMATO—a collaborative spam filter system [Bur04] designed and implemented by Nicolas Burri. During six months, SPAMATO has been improved, extended and redesigned. Some of these changes or additions are rather big and important, some are smaller but nonetheless useful, and some are just "nice to have".

In this chapter, Burri's initial SPAMATO system is summarized. Thereafter, most of the design and implementation changes and extensions are briefly described. The most important extensions—the "Razor Filter", the "Authorization System", and the "Trust System"—are more thoroughly discussed in separate chapters. See Chapters 3, 4 and 5 for details on these topics.

2.1 Initial Spamato System

The initial system consists of several parts and aspects which are described in the following sections. An overview of the structure of the initial system is shown in Figure 2.1.

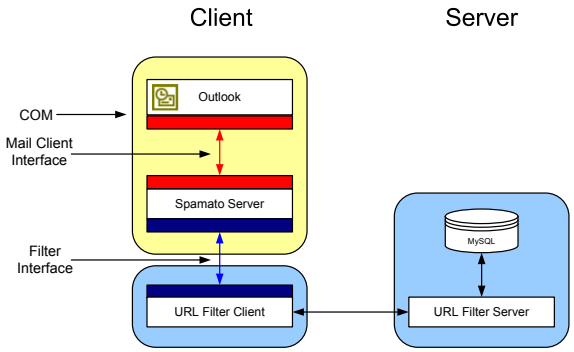

Figure 2.1: The setup of the original SPAMATO system (taken from [Bur04]).

2.1.1 Spamato Framework

The initial SPAMATO System has been designed as a spam filter framework (entitled as "Spamato Server" in Figure 2.1). Users of the system classify mail messages as "spam" or "not spam". SPAMATO takes care of this classification by trying to remove mail messages that have been classified as "spam" from as

many other users' mail boxes as possible.
The framework is implemented using Java [Sun94], which renders it platform independent.

2.1.2 URL Filter

The framework is able to handle any arbitrary number of spam filters at the same time. This is achieved by providing an interface for the common functionality of all filters that can be used within SPAMATO. This ability, however, had not been thoroughly tested since, so far, only one filter had been implemented: the *URL Filter* (consisting of the "URL Filter Client" and "URL Filter Server" in Figure 2.1). This filter, implemented in Java, parses mail messages for URLs and uses them to calculate an identifier (fingerprint) for each mail message. The fingerprint is reported to a server. If the same mail message arrives in another mail box, the URL Filter checks if this message has already been classified as "spam" on the server and, if so, removes it from the mail box.

The filter also provides the ability to do some basic configurations—for example to specify the server to connect to and the username and password to identify the user at the server (Figure 2.2).

The server side is designed as a Java component using a MySQL database.

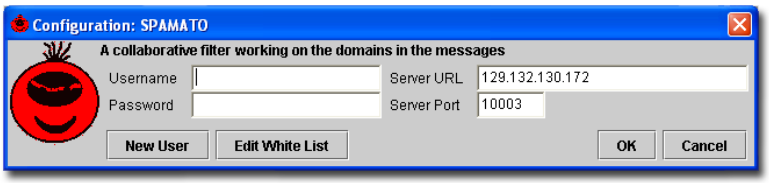

Figure 2.2: The old user interface to configure SPAMATO or the URL Filter, respectively (taken from [Bur04]).

2.1.3 Microsoft Outlook Add-In

A simple COM add-in (Figure 2.3) has been designed and implemented to enable the use of SPAMATO within Microsoft Outlook by simply clicking buttons on a command bar. This Add-In is entitled as "Outlook" in Figure 2.1. Most of SPAMATO's functionality, however, is encapsulated inside the SPAMATO framework. The add-in only delivers mail messages to this framework and moves them to a dedicated mail folder depending on the answers it receives from SPAMATO. It particularly watches the main "Inbox" in Outlook for incoming messages and delivers them to SPAMATO.

The add-in is implemented in Visual Basic .NET. Therefore it cannot call the Java-based framework by calling methods, but has to communicate with it through TCP sockets using XML messages.

Figure 2.3: The old user interface to control SPAMATO within Outlook (taken from [Bur04]).

2.1.4 Summary

The initial SPAMATO is built on two components: the framework and the URL Filter. An Outlook add-in has been implemented to use SPAMATO in Outlook. Users can vote for messages as "spam" (also known as *report*) or "not spam" (known as *revoke*). If the reports for a mail message reach a certain level, the message is filtered out for all users who receive this message thereafter.

2.2 Extensions and Improvements

In the following sections, all changes to the described initial system will be described. More information about the three most important extensions is provided in Chapters 3, 4 and 5.

2.2.1 Spamato Framework

The SPAMATO Framework is designed as a middleware between a set of spam filters and one or more mail clients. This Framework has been redesigned to satisfy the needs that appear if more than one specific mail client and more than one spam filter should be supported.

Local Server

Originally, SPAMATO acts as a local server listening on TCP port 7998. This is a perfect way to provide a possiblity to use SPAMATO in Outlook because the add-in (written in Visual Basic .Net) cannot call methods (written in Java) directly. If, however, another mail client can be used, such as Mozilla [Moz04], it is possible to call Java methods directly from code. SPAMATO has been redesigned such that it is possible to use it without a local server running but by calling methods on the main SPAMATO class directly.

Statistics

An interesting feature has been added to the SPAMATO framework: It maintains statistics about each spam filter used in SPAMATO and about the system itself. This is done by logging the checks of mail messages (and the results of these checks) and the reports and revokes of the messages. Example statistics are shown in 2.4.

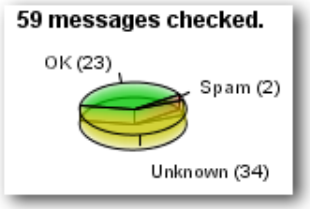

Figure 2.4: The statistics for one of the spam filters used by Spamato.

Filter Interface

Some parts of the interface of spam filters in the Spamato framework have been redesigned. Filters can now provide GUI panels to modify their settings. Another extension has been introduced wich enables filters to receive mail messages containing configuration data.

Razor/SpamNet Filter

In addition to the existing URL Filter, another spam filter has been integrated. This filter calculates a hash value using the body of a mail message and queries a database if this hash value is already known as spam. Adding this filter enables Spamato to work based on a much bigger voting user community thus increasing the accuracy of the whole system. Details about this filter are given in Chapter 3. This chapter also describes the process of how to add a new spam filter to Spamato.

Miscellaneous

As soon as Spamato included more than one spam filter, the requirement appeared to have a more sophisticated way to let users configure the system and its filters. A configuration dialog has been designed which is easily extendable for any number of settings (see Figure 2.5). Filters can provide GUI panels[1] that allow modification of their settings. This is done by the Spamato framework by calling the method `getConfigurationPanels()` of all filters. When the configuration dialog is closed, the system informs all filters of changed settings by calling their `saveSettings(configurationPanels)` method if the user clicked the OK-button.

Another feature has been added to notify the user: If Spamato identifies a mail message as spam, it plays a sound file to inform the user that a spam message has been detected. The sound clip can be chosen in a configuration dialog (as shown in Figure 2.5). This sound clip informs the user about incoming spam message so that he does not have to interrupt his work.

[1] These panels can be organized hierarchically.

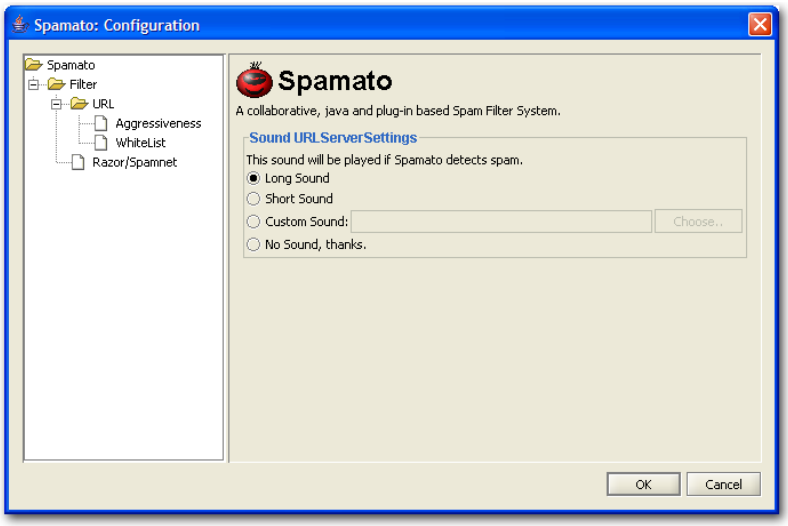

Figure 2.5: The Dialog for changing all settings of the SPAMATO system and all spam filters used by SPAMATO.

2.2.2 URL Filter

First of all, the URL Filter has been adapted to the changed spam filter interface provided by the SPAMATO framework. These adaptations but also some revealed bugs lead to several improvements in the filter that are described in the following sections.

Persistent Settings

All settings stored in a file are not in binary format any more, but in plain text property files. These files hold information in the form `key = value`, making it more readable and independent from the implementation. This plain text format has also been used for new files which have been introduced in addition to the old ones. In cases where disk space may become an issue (large storage files) the corresponding property file is stored in a zipped version.

URL Whitelist

In the initial version the URL filter maintains a whitelist of URLs, describing a value of trust for all URLs that have appeared in mail messages. Trust values are automatically adapted if a message has been classified as spam by the URL Filter. The whitelist has now been improved to adjust these values based on the global decision of the whole SPAMATO system instead of the decision by the URL Filter only. The user interface to change the values for given URLs or to add or remove URLs from the whitelist has also been improved (Figure 2.6). As mentioned in the previous section the whitelist is stored as plain text into a zip file.

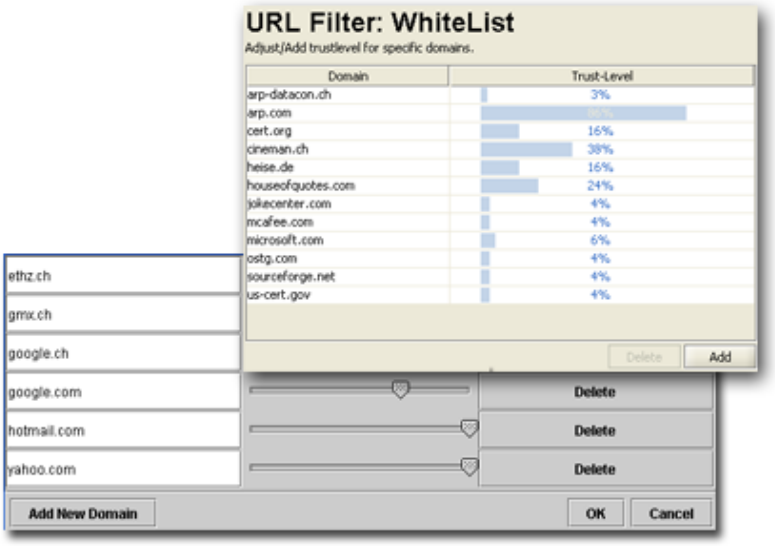

Figure 2.6: The old (in the background) and the new GUI (in the foreground) to edit and view the URL whitelist.

URL and Domain Handling

The URL Filter works with the URLs it can extract from mail messages. The following happens in the original URL Filter version. All URLs found in a email body are extracted using regular expressions. All these URLs are sent to the URL Server which extracts the domains from these URLs also using regular expressions. Then, all domains are concatenated and a hash value is calculated for this concatenated string. The database is queried for this hash and a count representing the number of reporters and/or revokers is returned to the client.

This procedure has been sharpened up in several aspects. First, the extraction of URLs has been improved to perform faster and more accurately by changing the regular expressions. These URLs are then sent to the URL Server as before. The server decodes all possible encodings of a URL to get a normalized string representation of the URL. This reduces the possibility to cheat the system by simply encoding one single letter in a domain name in hexadecimal format.[2] Then, the domains are extracted from the decoded URLs and duplicate domains are eliminated. The remaining, unique domains are alphabetically sorted, concatenated and hashed only now. This always leads to the exactly same hash value even if the order of the URLs in the mail message changes or a URL appears more than once in an email.

[2] As an example, the URL
http://%68%65rba%6Cm%65%64%73%6F%6E%6C%69%6Ee.com/alg/vedox
is decoded to
http://herbalmedsonline.com/alg/vedox.

Trust System

The question of whom to trust in a collaborative system is not easy to answer. The initial SPAMATO system did not try to find a way to prevent opponents from harming the collaborative decision but explicitly left this question for future work. A solution to this problem has been designed and implemented during this thesis. A detailed description of this solution is provided in Chapter 5.

Authentication and Authorization

Burri's URL Filter used an authentication based on username and password. A user has to create an account by specifying username and password. This data is transmitted to and stored on the server without any encryption. Communication with the server takes place by creating a communication session with the URL Server by sending username and password to the server. The whole authentication system has been redesigned. An overview of the new system is given in Chapter 4.

URL Server

The URL Server has been almost completely redesigned. It uses a new authentication and authorization system as well as a new trust system.

The server parts have been split into three main components: The voting system (delegated to the trust system), a statistical data collector, and the main system handling the user database. The database used by the URL Server has also been split into these parts and consistent naming of tables and columns has been introduced. The statistical component can be switched on or off when the server is started. Additionally, a configuration file mechanism has been included to configure the server parameters without having to change the source code. Furthermore, the identification of mail messages in the database storing the user votes has been decoupled from the whitelist. This means that a change in the set of domains in the whitelist does not imply loss of all previously stored data in the database.[3]

Finally, the communication between URL Client and URL Server has been changed to use XML messages, making the communication indipendent of version differences between client and server.

2.2.3 Microsoft Outlook Add-In

Functionality

The initial SPAMATO System checks the main inbox (named "Inbox") of Outlook only for incoming messages. If IMAP mail accounts are used, the incoming

[3]The original system uses a hash value of all domains of a message which are not on the whitelist to identify the message. If a domain is added to or removed from the whitelist, all messages containing this domain are useless for future requests or votes because their identifying hash value has changed. If the identifier does not respect the whitelist, however, this information is not lost.

mail messages are never added to the main inbox and thus not checked (and especially spam messages are not filtered out) by Spamato.

This problem has been solved by allowing the user to specify any mail folder in Outlook to be monitored by the Add-In. All unread mail messages that are added to these folders—including newly incoming messages even in IMAP folders—are delivered to Spamato. If Spamato identifies a message as *spam*, it can be moved to the local "spamato"-folder.[4]

Communication with Spamato

Originally the add-in sends the contents of mail messages encoded in XML to Spamato and receives a string representing the answer. The XML messages contain exactly the number of tags to hold the information of a mail message. All tags are always included in the XML message even if the tags are empty. The fact that answers by Spamato are only unformatted strings instead of XML messages is certainly due to lack of time while implementing the initial version. These properties have been improved:

- The component producing these XML messages has been redesigned to be more flexible. It is now possible to encode any number of any kind of data into a XML message and send it to Spamato.
- Messages sent by Spamato to the add-in are now encoded in XML too.

These two improvements are used to send commands (that are not mail messages) to Spamato and to request information.[5]

User Interface

Compared to the original user interface (Figure 2.3), the new version (Figure 2.7) features many small inprovements.

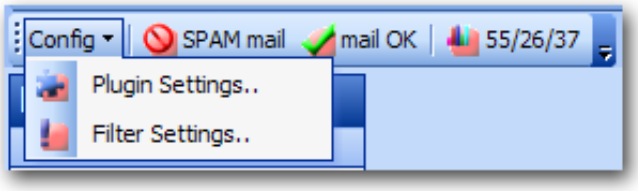

Figure 2.7: The new user interface to control Spamato within Outlook.

First, the add-in now uses its own command bar in Outlook and does not add its buttons to the standard command bar. The location of this new command bar is modifiable by the user. If Outlook is closed and restarted, the bar appears at the same place as it was when closing Outlook.

[4] Spamato moves messages that are identified as spam into a folder named "spamato" on the local machine. If the message origin is an IMAP server, it is downloaded to the local machine (into the "spamato"-folder) and tagged as *deleted* on the IMAP server.

[5] Currently Spamato is polled for statistical data by the Outlook add-in.

Second, there is now a submenu for changing settings (leftmost in Figure 2.7). The button originally named "Spamato" is now called "Config" and contains a menu holding two buttons. These Buttons each show a dialog to change the settings of SPAMATO or the add-in itself.

Third, a button showing summarizing statistics has been added (the rightmost button in Figure 2.7). The numbers mean from left to right how many messages have been checked so far, the number of messages that have been identified as spam, and the number of messages the user reported to SPAMATO.

Fourth, icons have been designed to be displayed next to the button captions. This has been done to provide a consistent look of SPAMATO indipendently of what mail clients may be supported in the future.

Miscellaneous

In addition to the extensions and improvements mentioned above, the add-in is now able to work in special cases when, for example, Outlook is not fully started. Sending a mail message by right-clicking a file and choosing SEND TO > MAIL RECIPIENT opens an email editing window only, without starting Outlook completely. The add-in does not crash any more in such cases.[6]

Finally, the add-in now correctly stores every user's settings into the appropriate folder, thus enabling spamato to be used by more than one user on a workstation—and these users do not need to have administrator privileges anymore. The location of this folder is also passed to permit SPAMATO to store its data in the same place.

[6] This is a case that not even commercial add-ins handle correctly. For example taking the described action always leads to a crash of Outlook if you use Cloudmark's "SpamNet" [Clo00] (version 2.41).

3 Razor/SpamNet Spam Mail Filter

One of the main problems of the initial version of SPAMATO is its very limited number of users. The performance of a collaborative system highly depends on the quality and size of its spam database. If no one reports mail messages, no messages are recognized as spam. Increasing the number of users and thereby the number of voters is one of the big needs to improve the accuracy of SPAMATO.

Since it is not easy to find beta testers and increase the number of users of SPAMATO, it has been decided to implement a new spam filter that queries the database of another (bigger) community of collaborative spam killers—the Razor/SpamNet community.

This chapter introduces Razor/SpamNet and its community. It then describes how an additional spam filter can be added to SPAMATO and how the Razor/Spamnet Filter has been implemented.

3.1 Razor/SpamNet and its Community

Razor [Pra99] is an open source[1] spam filter for unix systems implemented in Perl. It is based on the collaboration of Razor's users: A user reports a mail message that he identified as spam or revokes a message that has been marked as spam but in fact is a legitimate email. If a user receives a message, this message is checked if it is spam by connecting to a server and querying if other users have submitted reportes and/or revokes for the same mail message.

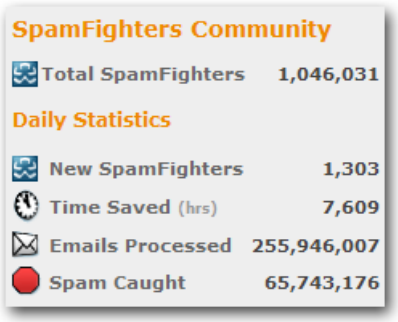

Figure 3.1: The SpamNet community in numbers (taken from the cloudmark webpage [Clo00]).

[1] The client part of razor is open source—the server, however, is not.

Spamnet [Clo00] is a commercial version of Razor for the Microsoft Windows users developed by the inventors of Razor. It is *not* open source and has been designed as a Microsoft Outlook and Outlook Express add-in only. The network and database used is the same as Razor's. There are two main differences between the two systems. First, SpamNet uses more algorithms to hash a mail message than Razor does and therefore achieves a higher spam detection rate. Second, the community of SpamNet is very big compared to the one of Razor. Some facts about SpamNet are shown in Figure 3.1.

Because of the similarity of Razor and SpamNet they are treated as *one* network and product in this thesis.

3.1.1 The Functional Principle of Razor/SpamNet

The classification of a message as spam or not spam is a process of several steps in the Razor network. This process is presented in the following sections.

Discovering a Set of Razor Servers

Razor is a client/server-based collaborative spam filter system. A client has to regularly discover the available servers to check and report or revoke mail messages. The following steps are performed by the client to build a list of available servers:

1. A set of *Discovery Servers* (D) is collected using the Domain Name System (DNS): The client tries to resolve the IP addresses of domains of the form X.razor2.cloudmark.com, X being a character of the list $(a, b, c, \ldots, z)$. The characters are used in ascending order and the resolution stops as soon as address resolution fails, indicating that no more Discovery Servers are available. All resolved IP addresses build the set D.

2. The client connects to a server in D. If the connection fails, a connection to another Discovery Server of the set is established. If no connection can be established at all, the system fails.

3. The client requests a set of *Catalogue Servers* C and *Nomination Servers* N. Catalogue Servers are used to check if a message is spam; Nomination Servers receive reports and revokes for messages.[2]

4. The servers of C and N are sorted in ascending order according to the Round Trip Time (RTT) from the client to these servers.

D, C and N are cached and stored persistently such that the above discovery process can be reduced to once a week or similar.

[2] It is undocumented how the database between these two types of servers are synchronized.

Checking a Message on the Razor/SpamNet Network

Razor calculates fingerprints for mail messages and queries a database if these fingerprints are known as spam messages. Its design allows any number of algorithms to calculate these fingerprints but only two are in use.[3] The first algorithm is named *Whiplash*. Whiplash uses URLs similar to Spamato's URL Filter but does not treat them as sets but rather takes every URL as an indicator by its own. This algorithm has only been recently added to Razor (at the time of writing this thesis) and is still producing too many false positives.[4] The second algorithm is the *Ephemeral* algorithm. Checking a mail message and generating a fingerprint in this algorithm includes the following steps:

1. The client connects to a server of the Catalogue Server set C.
2. The server sends a greeting to the client including a seeding nubmer s for its random number generator.
3. The client picks two random[5] parts of the mail message body.
4. These two parts are concatenated and a hash value using the SHA1 algorithm is calculated.
5. The client checks if this hash value is known to the server it is connected to.

3.2 Adding a Spam Filter to Spamato

Spamato is designed as an easily extendable software framework. Adding a new spam filter is therefore a simple task. A software component which should be used as a new filter in Spamato has to comply with the following requirements:

1. The implementation language must be Java.
2. It has to implement the interface `ch.ethz.common.ISpamFilter`.
3. In order to be loaded by Spamato it has to be manually added to the set of filters in `ch.ethz.common.main.FilterListUtil`.[6]

If all these requirements are fulfilled, Spamato will automatically use the component as an additional spam filter.

[3] SpamNet actually provides more than these two signature algorithms. However, since they are not open source and not documented, nothing can be said about them.
[4] A *false positive* is a mail message which is wrongly classified as spam.
[5] These positions are in fact not random at all because the random number generator is always seeded with the same seed s. This seeding always leads to the same sequence of random numbers.
[6] The loading mechanism should work in a more dynamic way in future versions of Spamato.

```
server: sn=C&srl=248&a=l&a=cg&ep4=7542-10

client: cn=Razor-Agents&cv=2.36
        a=g&pm=state

server: -sv=3.36
        sn=C
        zone=razor2.cloudmark.com
          //.. more status code
        ep4=7542-10
        ep8=5
        .

client: -a=c&s=?&e=?
        3csA8EHxF-yYraq7nqMrRMLTlx0A,4
        yDknCV8zQo-5Swlw2WyDBQn08c4A,4
        Xwc0s_V6IGcR3v1y3BF3gNxzd6oA,4
        yDknCV8zQo-5Swlw2WyDBQn08c4A,4
        .

server: -p=0
        p=0
        p=1&cf=100
        p=0
        .

client: a=q
```

Listing 3.1: A logged TCP stream: Razor checks four mail messages if they are known as spam to the server. One of them actually is.

3.3 Implementing a Java Client for Razor

The Razor/SpamNet Filter used in SPAMATO is based on version 2.36 of the original open source Perl implementation of Razor Agent. As mentioned in previous sections, only the client side of Razor is open source, therefore almost nothing is known about the server side. Unfortunately, this lack of information also includes the communication protocol between client and server—all that is known about the protocol has been obtained from guesses [Ste02] and from reverse engineering, for example by logging TCP streams (Listing 3.1) while Razor is running.

3.3.1 Overview

The implementation of the Razor/SpamNet Filter consists of six main components:

Discovery. This component handles everything to find sets of *Discovery*, *Catalogue* and *Nomination Servers*.

Connection. Everything concerning connections to any kind of server is encapsulated in this component.

Preprocessor. Razor handles messages in source code format as they appear in a "mbox" file on a Unix-like operating system. If this source code is unavailable (as for example in a message received from Outlook) it has to be rebuilt. Any special encodings are decoded, such as Base64, HTML and similar. The preprocessor takes care of these operations.
Since the Java version of Razor should be operating system independent, it is also necessary to take care of the different file formats.[7]

Hashing. All preprocessed messages are hashed to a fingerprint using the *Ephemeral* algorithm.

Agent. This is the main component. It controls the data flow between all other components and implements the interface `ISpamFilter` to be used as a spam filter from SPAMATO.

3.3.2 Problems and Challenges

Razor could not easily be ported to Java because several problems made the task more difficult than expected. Some of the problems and their solutions are described below.

Random Number Generator

The *Ephemeral* Algorithm picks some sections of the email body randomly in order to calculate a fingerprint for the message. These sections are, in fact, not random because the server defines a seeding for the client's random number generator (RNG) before the algorithm is executed. By providing this seeding, the same random number sequence results on every Unix and Linux machine running Razor. Therefore, the Java port of Razor must calculate the same random number sequence as its Perl equivalent. This is necessary in order to pick the same "random" sections of mail messages and calculate identical fingerprints.

In the original Razor implementation Perl's built-in RNG `rand()` is used. This call is internally mapped to the native RNG `drand48()` of the *GNU C Library* [GNU96] also known as `libc`. It was therefore necessary to port this RNG from C to Java—a very challenging task because of two facts:

Library-based implementation. Code is very heavily reused in `libc`. There is not only one single file containing the implementation for the RNG, but it is distributed throughout the library. Extracting the code needed for the RNG was therefore somewhat cumbersome.

Number representation-based implementation. `libc`'s RNG relies on shifting bits directly in the memory representation of floating point numbers standardized in IEEE 754 [Ins85]. To also be able to perform this bit shifting a reduced version of IEEE 754 had to be implemented in java.

[7] The line terminator in Unix and Windows is an example for such a difference.

Interoperability

Razor, being designed for Unix-like systems only, does not have to deal with cross-platform compatibility. A Java implementation however, has to respect these differences. A spam filter in SPAMATO additionally has to be able to work with different mail clients and thus different mail formats. Two aspects have shown necessary to be handled by the Java implementation of Razor:

- All files have to be transformed to one single file format: the Unix file format. If this is not done, emails may contain more characters than they would on Unix and therefore result in different fingerprints.
- Not all mail clients store the original source code of incoming mail but rather in a proprietary format dropping the source code. A Java implementation has to deal with this fact and has to try to rebuild "some kind of" source code in order to calculate fingerprints correctly.

3.3.3 Unsolved Tasks and Problems

Although the Java implementation of Razor works quite well, there are some problems and tasks that have not been solved during this thesis:

Pinging Servers. Razor sorts its Catalogue and Nomination Server lists according to ping times. However, it does not use ICMP pings but connects to the servers' echo port using TCP in order to circumvent firewalls. Connecting to the echo port on any of this servers failed in tests. A solution to this problem has not been found by now.

Reports and Revokes. The Java implementation is only able to check if a message is known as spam in the Razor network . It is not able to report or revoke messages so far. This ability should be added as soon as possible to also contribute to the network instead of profiting only.

My identity and value is in direct proportion to what I see in others.

Horacio Antunes

4 AAAS: Automatic Authorization and Authentication System

The main goal of the Automatic Authorization and Authentication System (AAAS) is to provide an automatic way of registering users to a system in a secure way. Unlike many other systems, not an authentication mechanism based on username and password is used, but a mechanism based on public key encryption, digital signatures, and email.

Accounts are automatically created in AAAS. They are highly transportable, providing the ability to use the same account on several installations even at the same time. The transportation of an account to another machine is also accomplished automatically.

AAAS is used within the URL Filter of SPAMATO to let users create an account automatically. Only users having an account can submit reports and/or revokes to the URL Filter server.

4.1 Registering a New Account in AAAS

Only a valid email address is needed to register a new account in AAAS. As long as SPAMATO is controlled by an add-in of a mail client, this address can be automatically extracted from the mail client and delivered to AAAS.

The steps to register a new account are performed automatically, especially no user action is required. There is no password to be remembered and no username to be invented—everything is done in the background.

Registering a new account includes the following steps (see Figure 4.1 for a more schematical description of the protocol):

1. The server calculates a RSA key set R or reuses one that has been calculated before.
   The client calculates a DSA key set D.[1]

2. As soon as the email address is known to the client, a registration request is sent to the server. This request includes the client email address.

3. The answer of the server contains its public key in R. The client uses this public key to encrypt its own key set D. The encrypted D is sent to the server.

[1] DSA stands for "Digital Signature Algorithm". DSA keys are only used to create signatures. RSA is an asymmetric encryption algorithm. The name is derrived from its developers (Rivest, Shamir, and Adleman). RSA is used to asymmetrically encrypt data. This process is expensive and therefore only used when necessary.

4. The received encrypted key set D is decrypted by the server. A random challenge string c is computed and sent to the client email address. c and D are temporarily stored together with the client email address.

5. As soon as the email message containing c is received, the client appends a signature to it using the private key in D and sends it to the server.

6. The server verifies the signature using the public key in D and then compares the received c to the original one. If both tests are passed, the email address and the key set D are marked as confirmed and are not temporary any more. The client has successfully created a new account.

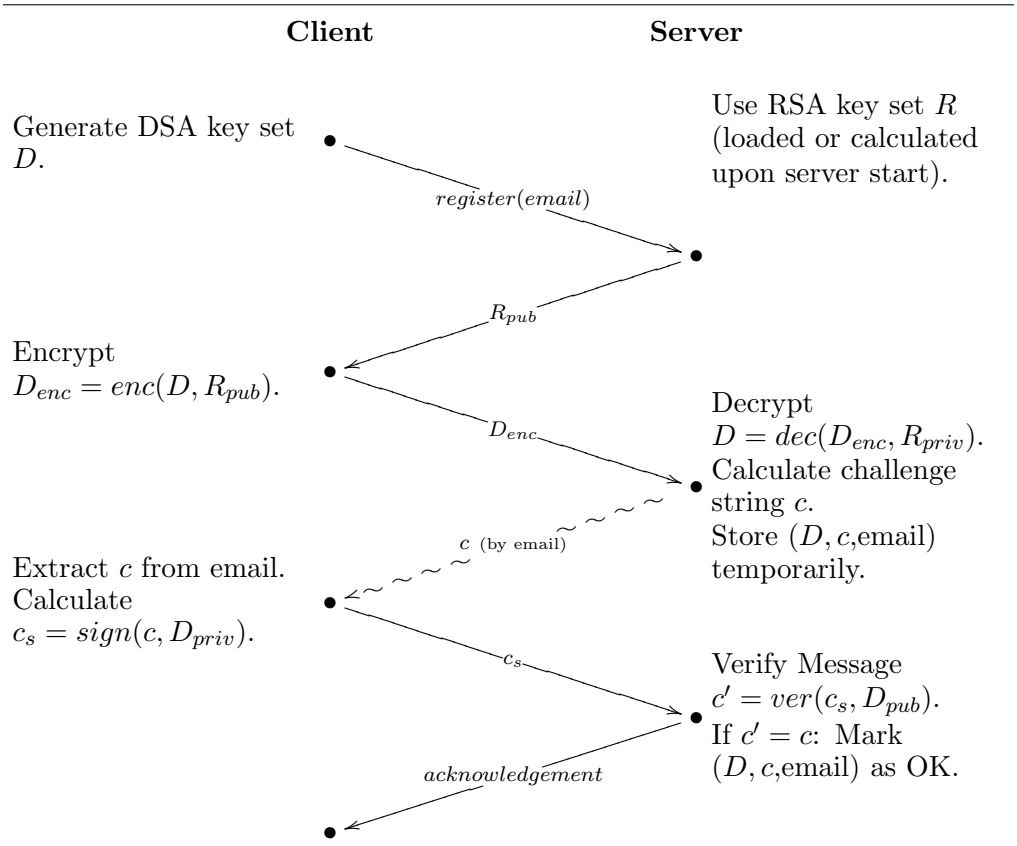

Figure 4.1: The process of registering a new account in AAAS. The functions $enc(x, y)$, $dec(x, y)$, $sign(x, y)$, and $ver(x, y)$ encrypt, decrypt, sign, and verify x using key y.

Using the above protocol implies verifying the email address of the client. The registration process cannot be finalized if a client does not provide a valid email address. In such cases, the account information stored by the server remains temporary and can be deleted after the expiration of a predefined period of time.

Successful termination of the registration protocol results in a client having a key set D and the server having a valid email address and a key set D of the client. Important messages from the client to the server can thereafter be signed by the client using the private key and validated by the server using the public key in D.

4.2 Reregistering to an Existing Account

The protocol to register a new account provides the possibility to register to this account from another computer or to reregister to it after reinstallation. In the following, both actions are referredto as *reregistering*.

A client needs its DSA key set D to sign messages it sends to the server. The server validates the identity of the client by verifying these signatures. Reregistering to an existing account can therefore only succeed if the client knows D, which is stored on the server, after the reregistration succeeded.

The protocol to exchange D and therefore reregister to an existing account in AAAS is designed as follows (for a more schematical description of the protocol see Figure 4.2):

1. As soon as the email address is known to the client, a temporary RSA key set R is created in order to be able to receive encrypted messages. The public key in R together with the email address is sent to the server.

2. The server performs a lookup to its database. If the received email address is stored in the database, the DSA key set D of this user is extracted, encrypted using the public key in R, and sent by email to the provided email address.

3. When the email containing the encrypted key set D arrives, the client decrypts D using the private key in R, stores D, and drops R.

After this protocol has finished successfully, the client is in the same state as if this account had been newly registered. Only a user having access to the email address of this account can reregister.

4.3 AAAS in Spamato

The URL Filter of Spamato uses AAAS to register users automatically. The Spamato framework receives the email address of the user from the add-in controlling Spamato—for example from the Outlook add-in. All spam filters get this information from the framework. When the URL Filter is in its unregistered state, obtaining this email address initiates the AAAS registration or, if this email address is already registered, the reregistration process.

Both processes are split into two parts—the part before the email is sent by the server, and the part after the email is received by the client. When the first part of the respective process is finished successfully, the URL Filter enters an intermediate state awaiting the mail message from the URL Server.

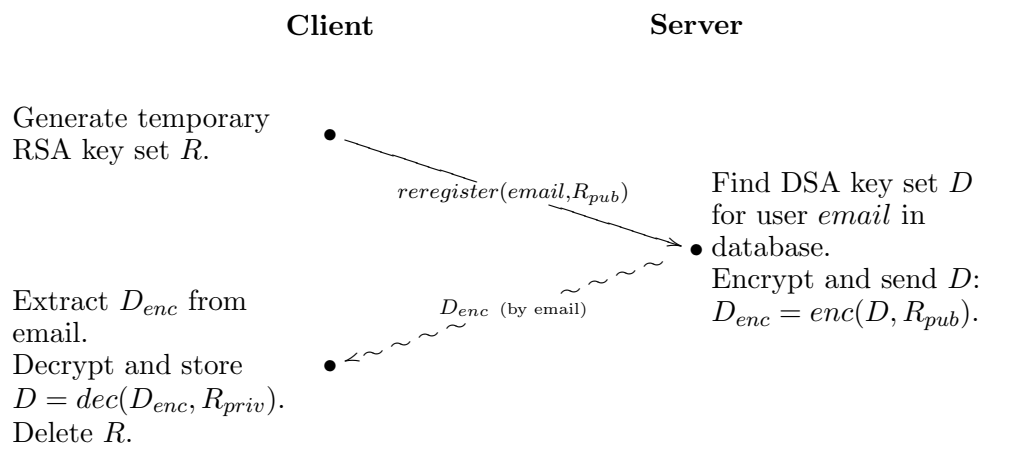

Figure 4.2: The process of reregistering to an existing account in AAAS. The key task of this protocol is assuring the identity of the client and transmitting the DSA key set D to the client in a secure way. The functions $enc(x, y)$ and $dec(x, y)$ encrypt and decrypt x using key y.

If the URL Filter is in this intermediate state, obtaining the expected email message initiates the second part of the respective process. The message is then declared as a configuration message and is not classified by the framework or any spam filter.[2] Receiving such an email message while not in intermediate state does not affect the URL Filter. It is treated as if a normal mail message arrived.

4.3.1 Using the Obtained Key Set

The key set resulting from the AAAS protocol is not used for all communication sessions. The list below describes when they are used and when not:

Checking mail messages. The URL Filter checks mail messages by submitting the contained URLs to the URL Server. Since this is by far the most frequent action, checks do not use the key set in order to perform as fast as possible.

Reporting and Revoking mail messages. The Trust System (Chapter 5) punishes or rewards users who submit false or proper votes, respectively. Therefore, it is necessary to be sure of who submitted which reports or revokes. The URL Filter takes care of this requirement by signing report and revoke messages with the user's private key obtained by means of the AAAS protocol.

[2] The SPAMATO framework delivers incoming mail to all spam filters. If one spam filter declares a mail message as a configuration message, it is ignored by the framework and no classification of the message is performed.

A day can press down all human things, and a day can raise them up.
But the gods embrace men of sense and abhor the evil.

Athena to Odysseus

5 Trooth: An Advanced Trust System for Collaborative Voting

Any collaborative spam filter system highly depends on the reports and revokes of its users. The main goal is to need as few votes as possible while guaranteeing a correct classification of messages for as many users as possible. Ideally, if all users classify all messages identically, no errors happen, and nobody is trying to cheat the system, one single report suffices to remove this spam message from all other inboxes.

In reality, however, not all users classify mail messages identically and, since users are human, they make mistakes. The collaborative system must therefore provide a way to deal with conflicting and erroneous votings. Furthermore, there are malicious users: people who send spam mail messages (spammers) are keenly interested in avoiding or sabotaging spam filter systems to get their spam through them or render them useless for others by forcing false positives. A stable collaborative system should barely be affected by malicious votings as long as they are a minority.

Summarizing, any collaborative system has to deal with votings that can be classified into the following groups:

Proper votings. If the majority of voting users agree (meaning that almost only reports or revokes are submitted), there is no need for the system to take part in the voting process.
In a collaborative spam filter, such votings concern messages which are clearly classifiable.

Controversial votings. No clear majority of reporters or revokers is achieved. Ideally, the system deals with this case by deciding according to each user's individual assessment.
These votings typically appear when users receive newsletters by email. Some do not want this newsletter and therefore report it as spam although this is not really spam if considered objectively.

Erroneous votings. It may happen that users accidentally click on the report button submitting a vote they did not intend.
For the voting process of a message these votes may be neglected since the number of erroneous votes is usually small. The system only has to deal with reported private mail messages, for example by dropping such messages if the user is the only one who reports *and* revokes a message.

Malicious votings. Each system has its enemies. There are users who want to break or cheat the system in order to influence the voting process. A robust collaborative system should be almost immune to users who try

to break it by submitting wrong reports and revokes—at least as long as the enemies of the system are a minority.
Malicious votes are submitted mostly by spammers who either want their spam to get through the spam filter or to enforce false positives rendering the spam filter useless to most users.

Most of today's trust systems are able to handle three of these groups: proper, erroneous and malicous votings. An explanation of how this is done is given in the next section. None of these systems, however, are able to handle controversial votings in a satisfactory way. The following sections, therefore, introduce Trooth. Trooth is an advanced trust system assumin a new approach to handling controversial votings. It processes such votings for every user individually by accounting for her prior reports and revokes to classify new incoming messages.

5.1 Standard Trust System: Additive Increase and Multiplicative Decrease

The most widely used approach to a trust system is the *additive increase and multiplicative decrease* (AIMD) method. AIMD assumes the existence of a globally valid classification of mail messages. A trust value $v_t(u)$ is assigned to every user u, representing her worthiness for the classification process of the collaborative system.

Any message m is categorized either into the category $\mathbf{S}$ of spam messages or in category $\overline{\mathbf{S}}$ of legitimate messages—no other category is possible. The global classification of m depends on the accumulated trust value of reporters and revokers. A message is declared as spam if the accumulated trust value of the reports is higher than the value of the revokers ($\mathbf{P}$ being the set of reporters and $\mathbf{V}$ the set of revokers of m):

$$m \in \mathbf{S} \quad \Leftrightarrow \quad \sum_{p \in \mathbf{P}} v_t(p) > \sum_{v \in \mathbf{V}} v_t(v). \tag{5.1}$$

Depending on this global decision, reporters and revokers are rewared or punished, respectively. If m is classified as spam, all trust values v_t of reporters are increased while the trust values of all revokers are decreased ($k \in N^+, f \in R_0^+, f < 1$):

$$\forall p \in P: \quad v_t(p) \;=\; v_t(p) + k \tag{5.2}$$

$$\forall v \in V: \quad v_t(v) \;=\; v_t(v) \cdot f. \tag{5.3}$$

If m is classified as a legitimate message, everything is calculated the other way round: The revokers are rewarded while the reporters are punished.

5.1.1 Advantages

As showed in Equations 5.2 and 5.3, increasing the trust value v_t is much more difficult (additive) than decreasing it (multiplicative). As long as malicious

users are a minority, their trust value is decreased rapidly, reducing their influence to the global decision vote by vote. Honest users, however, slowly gain more and more influence to the decision. The higher the trust value of honest users, the fewer are necessary to categorize a mail message correctly. AIMD therefore works very well if a global categorization of a message is possible.

5.1.2 Disadvantages

The assumption of AIMD, that all mail messages can be globally classified as spam or legitimate messages, is just not true. There are people who want to buy medicals on the internet and therefore do not classify such messages as spam. There are also people who just do not want to receive that AOL newsletter every month and (wrongly) report it as spam. If a message cannot be globally classified, AIMD fails for a not necessarily small minority which does not share the classification of the majority. Furthermore, the details of how exactly trust values are decreased and increased have to be kept from public knowledge in order to prevent users from acting in a way that significantly harms the system.[1]

5.2 The Trooth Approach

Trooth solves the disadvantages of AIMD while at the same time being more robust. It allows the server side to be published without any restrictions. To achieve this, the four groups of votings a collaborative system has to deal with (see at the beginning of this chapter) are regrouped into only two groups:

Normal votings. This group is the same as "proper votings" in the introduction of this chapter. It contains votings which have a clear majority of either reporters or revokers allowing a clear classification of the respective messages.

Special votings. It contains the remaining groups: controversial, erroneous and malicious votings. Due to the design of Trooth, these three can be handled in exactly the same way.

If a voting has to be treated as normal or special is decided by the server component based on two parameters:

1. the number of voters for a given message and
2. the proportion of the number of reporters and revokers.

The next sections describe the functionality of Trooth in more detail.

5.2.1 Handling of Normal Votings

If the server decides that a voting results in a clear majority with only few dissenting votes, the mechanism for normal votings (in the following referred to as *normal case*) is executed. This is by far the more common case than the one for special votings (referred to as *special case*) as indicated by their names.

[1] This is the reason why Razor's server side is not published.

Reporting and Revoking Process

Reporting or revoking a message is a very simple task if the voting is treated as normal. Reports and revokes are handled in exactly the same way but stored in different locations on the server:

1. The client c sends a report or a revoke for message m to the server.
2. The server adds c to the list of reporters $\mathbf{P}(m)$ or revokers $\mathbf{V}(m)$ of m. Only one report and one revoke per client and message is accepted,[2] therefore the lists $\mathbf{P}(m)$ and $\mathbf{V}(m)$ are sets.

Classification Process

In the normal case, the classification of a mail message m takes the following steps (assuming the server already decided the voting to be handled as a normal case):

1. The server calculates the number of reporters $|\mathbf{P}(m)|$ and revokers $|\mathbf{V}(m)|$. Then, it sends these numbers to the client.
2. The client classifies the message according to $|\mathbf{P}(m)|$ and $|\mathbf{V}(m)|$.

The differences to AIMD are obvious:

- Since the server simply counts the number of votes for a message, each user has the same value in the system. (In AIMD, each user has a trust value which may differ from the trust value of all others.)
- The classification happens *at the client*, not at the server.

Advantages

Since each vote has the same value, the server does not have to take care of the trustworthiness of each user. Moreover, no user is preferred, even new users are able to influence the result of a normal voting.

Moving the classification from the server to the client introduces new possibilities. Clients can modify parameters of how the classification is calculated. Such parameters include:

- The total number of voters.
  Only considering votes containing more than x opinions, for example, renders the result more reliable. This number can be very low if the voters are trustworthy, it has to be increased if more and more malicious users vote.
- The proportion p_n of reporters and revokers.
  An intuitive proportion, as an example, is to classify messages as spam if more than 50.0% of the voters have reported it as spam. The lower this value, the more aggressive the filtering will be.

[2] If a client reports and revokes the same message, both votes can be deleted from the server.

Disadvantages

Obviously, this system can be very easily influenced by vicious users, for example by spammers. But in this case, the votings do not achieve a clear majority of reports or revokes and are therefore not treated as a normal but as a special case.

5.2.2 Handling of Special Votings

If no clear majority is achieved in a voting, the server decides to handle this voting as a special case, resulting in a more sophisticated but also more expensive process. As this process is only chosen under special circumstances, it is executed very rarely compared to the normal case, legitimizing to use more resources for its handling.

Reporting and Revoking Process

The reporting and revoking process in the special case is an extended version of the one in the normal case:

1. The client c sends a report or a revoke for message m to the server.
2. The server adds c to the list of reporters $\mathbf{P}(m)$ or revokers $\mathbf{V}(m)$ of m. Only one report and one revoke per client and message is accepted, therefore the lists $\mathbf{P}(m)$ and $\mathbf{V}(m)$ are sets.

So far, both processes are identical. The next steps, however, are only needed if the server handles the voting for m as a special case.

3. The server calculates a subset of reporters and revokers for m:
   - The identifications (IDs) of all reporters are mapped to a ring. To achieve this, the IDs are sorted in ascending or descending order. The last entry in the list refers to the first entry as its following entry. The first entry refers to the last one as its preceding entry.
   - The server selects a section of this ring containing the ID of the client c in its center. This results in a subset $\mathbf{P_c}(m)$ of $\mathbf{P}(m)$. Assuming that most of the time the same group of users receives the same messages, this subset is very similar for any m.
   - The subset $\mathbf{V_c}(m)$ is selected identically.
4. The subsets $\mathbf{P_c}(m)$ and $\mathbf{V_c}(m)$ are sent to the client.
5. The client handles these sets by storing who classified m according to his own vote and who classified it conversely. This is done by adapting[3] trust values for all user IDs in $\mathbf{P_c}(m)$ and $\mathbf{V_c}(m)$ in a trustlist $\mathbf{T_c}$. This adaptation can be done as in AIMD systems.

[3] Default values are assigned to unknown IDs before adapting their trust value. The adaptation is then calculated based on this default value.

The size of the chosen sections and therefore the size of the resulting subsets $\mathbf{P_c}(m)$ and $\mathbf{V_c}(m)$ is a configurable parameter. In a future version, it might be possible to adapt this parameter automatically for each user. How trust values are adapted is also potentially modifiable but rather by designers than by users of the system.

Classification Process

If the server decides to treat a voting as a special case, the following steps are performed in order to classify a message m (see also Figure 5.1):

Building list of reporters and revokers. The server queries the database for the list of reporters $\mathbf{P}(m)$ and revokers $\mathbf{V}(m)$ of the message m in question.

Calculating a subset of these lists. The subsets $\mathbf{P_c}(m)$ and $\mathbf{V_c}(m)$ of reporters and revokers are selected as described in the previous section.

Sending the lists to the client. $\mathbf{P_c}(m)$ and $\mathbf{V_c}(m)$ are sent to the client c.

Getting the most trustworthy votes of these lists. The *client* chooses the k most trustworthy votes from the received lists according to the trust values in its own trustlist $\mathbf{T_c}$. This results in the two sets $\mathbf{P'_c}(m)$ and $\mathbf{V'_c}(m)$ of cardinality k.

Calculating assessed votes. In order to classify m, the client calculates the accumulated trust value P of chosen reporters and revokers (V):

$$P = \sum_{p \in \mathbf{P'_c}(m)} trust(p) \tag{5.4}$$

$$V = \sum_{v \in \mathbf{V'_c}(m)} trust(v) \tag{5.5}$$

where $trust(x)$ is the trust value of x as stored in $\mathbf{T_c}$.

Classification. Finally, the classification takes place at the client. m is classified as spam if $\frac{P}{V} > p_s$, where p_s is a constant representing a value between 0% and 100%.

The aggressiveness of the classification process can be changed by adjusting the values of k and p_s.

Advantages

The system is very decentralized, leaving as much of the process to the client as possible. Therefore, the architecture of a system using TROOTH is much easier to be designed to work in a peer to peer network without a central server. The server, furthermore, does not have to handle any trust values since this is done by every client individually.

Using the described process for the handling of special cases, it is no longer necessary to differ between malicious users and those who classify messages

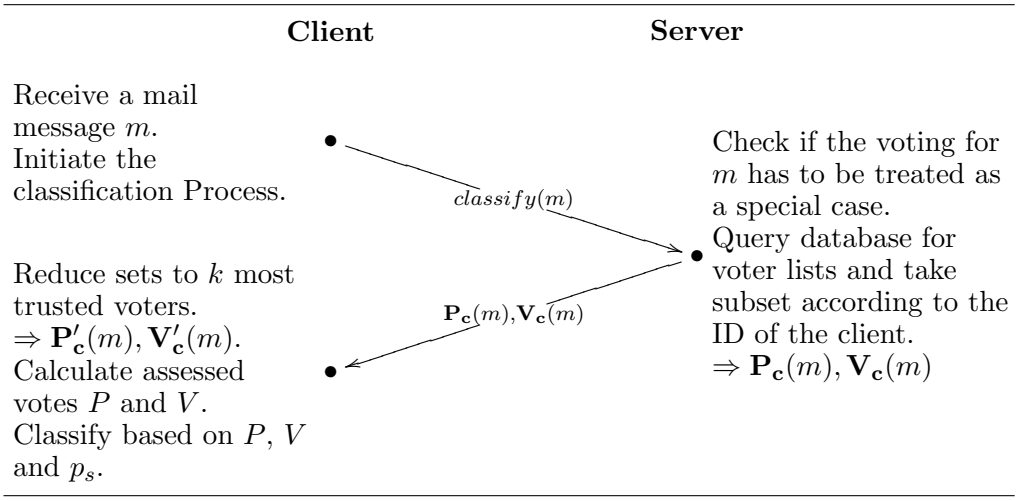

Figure 5.1: The process of how a mail message m is classified in TROOTH if the server handles the voting for m as a special case.

differently than oneself (either by error or by different interests). TROOTH treats them all as users who have different opinions by reducing their trust value locally at each user. Simplifying, TROOTH does not assume the existence of global decisions but the existence of user groups who classify the messages identically. By adapting the trust values locally, each user selects the members of his group himself.

Really harming the system is difficult:

- One user can only affect a small group of users in the system.
- To affect a user in his group, a malicious user has to become one of this user's k most trusted members of the group. But if this is the case, the malicious user rather helped than harmed the system.

Disadvantages

The special case produces much more traffic than AIMD and TROOTH's normal case because of the transmission of ID lists. These lists have to be stored on the client side. Currently, it is rather difficult to make predicitions about the size of such lists.

5.3 Implementation

TROOTH has been designed to be independent of the system it has originally been designed for. It is highly adaptable to any kind of collaborative system working with user identifications (user IDs) and fingerprints (IDs of what the users are voting for).

The system has been designed to provide a server and a client side which are both adaptable to the platform they are running on.

5.4 Trooth in Spamato

The URL Filter of Spamato uses Trooth to be able to classify messages, more precisely to discern spam from legitimate messages.

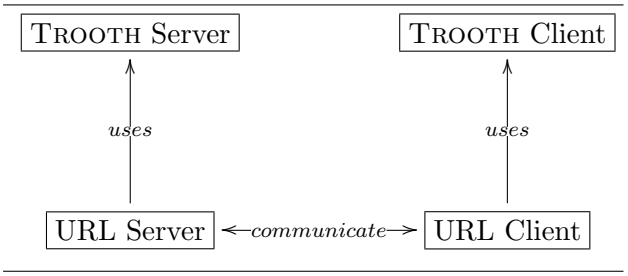

Figure 5.2: Trooth is used within the URL Filter in Spamato. The URL Server uses the server component of Trooth and sends the information to the URL Client. The client then uses the Trooth Client to classify emails as spam or legitimate.

The URL Server uses the server component of Trooth to handle votes for mail messages of its clients. It answers requests of the URL Client by sending an XML message containing the information which has been received from the Trooth Server. The URL Client retrieves these messages, extracts the information, and transforms it into a form the client component of Trooth is able to understand. It then hands this information over in order to classify the mail messages. The described structure is illustrated in Figure 5.2.

Users are allowed to change the settings x, k, p_n and p_s (as defined in Section 5.2) on the client side.

6 Summary

This thesis extends and improves SPAMATO—an existing, collaborative spam filter system. Such systems do not rely on algorithms or rules to identify spam but rather on humans. Users are connected to form a community whose members report messages they identify as spam. If a mail message is received, the community is queried if this message is known as spam and can eventually be removed from the inbox of the user. Since this approach highly depends on the trustworthiness of its participants, it is fundamental to authenticate all users in order to remember their reliability and honesty.

In this thesis, an authentication mechanism and a trust system fulfilling the above requirements are designed and implemented. Then, these systems are introduced to SPAMATO in order to protect it against malicious users. An additional spam filter is added to connect SPAMATO to a bigger community of spam fighters. SPAMATO is also improved in many other aspects, such as its user interface and its communication engine.

In order to increase the still small number of SPAMATO users, a new spam filter has been implemented: the *Razor/SpamNet Filter*. This filter uses the text-based algorithm *Ephemeral* to calculate fingerprints for messages. It then queries a big community of users if this fingerprint belongs to a known spam message.

Usually, community-based systems utilize authentication mechanisms based on username and password. In this work, a new approach is presented: the *Automatic Authorization and Authentication System (AAAS)* which does not require any user interaction. It is based on email addresses and key pairs. Registering a new account happens by simply generating a key pair at the client and submitting the email address to the server. If the client can prove having access to the account of the submitted email address, a new account in AAAS is assigned to the client.

Using AAAS, users are reliably identifiable which provides the possibility to rate their thrustworthiness. This way, malicious users can be prevented from harming collaborative systems. Most of today's trust systems, such as SpamNet's [Clo00] *TeS*, use the *additive increase, multiplicative decrease (AIMD)* approach. AIMD, however, fails in cases where no globally valid classification of a mail message exists. TROOTH, an advanced trust system, has been designed to achieve proper classifications even in these cases by handling messages according to the number of reports and revokes. If a clear majority of reporters or revokers exists, the message is treated as *normal case*, resulting in a voting where each user has one single vote. If no clear majority is achieved, the message is treated as *special case* in which users are separated into small groups. The message is classified at the client by either using the number of reporters and revokers (in the normal case) or the accumulated trustworthiness of the

most trusted reporters and revokers in the user's group (in the special case).

AAAS and Trooth, used within the URL Filter of Spamato, are working very well using the small beta tester community of Spamato. Benchmarking and simulating to verify their behaviour when many users have to be handled, however, remains to be done in future projects. This is out of the scope of this thesis but should be a topic in future work.

7 Future Work

During the design and implementation of the components described in this thesis, not all problems and questions could be solved and answered due to lack of time. Therefore, an overview of possible future tasks and open problems is presented in this chapter.

7.1 Razor/SpamNet Filter

7.1.1 Transmit Reports and Revokes to the Razor/SpamNet Network

The Razor/SpamNet filter is currently only ripping the Razor/SpamNet community off without contributing reports or revokes to it. In order to convince Razor or SpamNet members to use Spamato, contributing to their network is a necessary step. Furthermore, influencing the classification of the Razor/SpamNet Filter by transmitting reports and revokes would additionally improve the accuracy of Spamato.

7.1.2 Export Core as Open Source Project

Since Razor is developed under the GNU Public License GPL [GNU91], the Razor/SpamNet Filter must also be published as open source project. Therefore, the Spamato-specific code has to be separated from the Razor-specific core of the filter.

7.2 Automatic Authorization and Authentication System (AAAS)

7.2.1 Increase Performance

Currently, the transmission of the client's key set is encrypted using a free RSA encryption implementation. The performance of this implementation is very low, and replacing it with a faster implementation would improve the performance of the registration and reregistration process significantly.

7.2.2 Provide it as Component and /or Service within Spamato

AAAS is implemented to be used within the URL Filter. To render the system more easily reusable, it has to be designed as a more indipendent software component. Furthermore, by providing AAAS as a service in Spamato, which is accessible by every spam filter, it becomes more useful—no spam filter would have to implement an authentication system of its own.

7.3 Trooth

7.3.1 Simulation

In order to prove the effectiveness of Trooth, a big community including malicious users is needed. Since it is difficult to find enough users who are willing to test Spamato, a controlled simulation is a more probable option to test Trooth.

7.3.2 Adaption of Parameters

The parameters provided in Trooth—both at the server and at the client side—are currently guesses based on common sense. These paramters might need to be changed when the system has been running for some time and the simulation has been performed.

Bibliography

[Bur04] Nicolas Burri. Spamato: A collaborative spam filter system. Diploma thesis, Swiss Federal Institute of Technology Zurich, 2004.

[Clo00] Cloudmark. Spamnet. `http://www.cloudmark.com/spamnet`, 2000.

[GNU91] GNU Project. GNU General Public License. `http://www.gnu.org/copyleft/gpl.html`, 1991.

[GNU96] GNU Project. GNU C Library. `http://www.gnu.org/software/libc/libc.html`, 1996.

[Ins85] Institute of Electrical and Electronics Engineers. IEEE 754: Standard for binary floating-point arithmetic. `http://grouper.ieee.org/groups/754/`, 1985.

[Moz04] Mozilla Project. Mozilla Suite. `http://www.mozilla.org/products/mozilla1.x/`, 2004.

[Pra99] Vipul Ved Prakash. Vipul's Razor. `http://razor.sourceforge.net`, 1999.

[Ste02] William Stearns. Razor2 protocol. `http://www.stearns.org/razor-caching-proxy/razor2-protocol`, 2002.

[Sun94] Sun Microsystems. Java 2 Platform, Standard Edition. `http://java.sun.com/j2se/index.jsp`, 1994.

[UNs04] UNspam. Spam facts, numbers & statistics. `http://www.unspam.com/fight_spam/information/spamstats.html`, 2004.