Development, Deployment, and Rating of Plug-Ins

Technical Report TIK-259

Keno Albrecht, Roger Wattenhofer
Computer Engineering and Networks Laboratory
ETH Zurich, 8092 Zurich, Switzerland
{kenoa, wattenhofer}@tik.ee.ethz.ch

August 2006

Abstract

In this paper, we present a lightweight but powerful plug-in container which provides advanced features such as dynamic class loading, dependency, configuration, and security management. We highlight the deployment mechanism which allows to publish, install, and update plug-ins from arbitrary sources at runtime. Furthermore, we introduce the TROOTH voting and trust system which is used to assess plug-ins but has been designed as a general rating mechanism. Finally, we present the extensible architecture of the SPAMATO spam filter framework and show how it employs the plug-in mechanism and the TROOTH system in a real-world application.

1 Introduction

With increasing complexity, large software projects tend to get unwieldy, unmanageable, or even out of control. Researchers and developers have tried to counter this crisis with several approaches. Starting with modularization techniques such as abstract data types and object oriented programming, today, component and service orientation, plug-in architectures, and aspect oriented programming are "en vogue." The software development process evolves to a software management process, where more and more *building blocks* are developed separately and connected subsequently.

The building block or *black box* metaphor offers one chief advantage: a black box hides its actual implementation behind a well-defined *interface* facade which defines its functionality. In this paper, we survey a particular extended type of black boxes: *plug-ins.* More precisely, we examine frameworks which manage a bundle of plug-ins and their interdependencies. Traditionally, plug-ins have been used to extend the functionality of existing software. For instance, web browsers are extensible to support custom MIME types, such as Java or Flash, and imaging tools can support additional filters. Here, plug-ins are not deployed with the base product but integrated afterwards via a well-defined interface and extension mechanism. We, however, consider a more radical plug-in notion. In our framework, *everything* is a plug-in. Besides a small bootstrapper which initializes and connects the runtime system, the complete application logic is defined by cooperating plug-ins.

Open source projects such as the instant messaging client Gaim and the spam filter system SpamPal, which are still traditional frameworks, benefit from third-party developers who contribute additional features that can be plugged into the existing host application. But only in "everything-is-a-plug-in" frameworks such as Eclipse, the full power of plug-in development becomes apparent: not restricted to the default Java IDE, complete applications are built on top of the bare framework.

As the development of Eclipse has been driven to provide an application framework for building software, our plug-in framework has been shaped as we implemented the SPAMATO spam filter system. However, it proved to be useful in other projects as well.

For Spamato, we expect the development of several third-party spam filters and analyzing tools. Therefore, an architecture that allows for the seamless integration of such plug-ins is a must. Furthermore, we want a mechanism to publish, install, and update plug-ins at runtime, that is, without a restart of the application. This is particularly important for email filtering software since, otherwise, malicious messages can slip through and harm a user's machine. However, since plug-ins themselves come from unknown sources and, therefore, generally have to be suspected, a security mechanism has to prevent possible damages. Furthermore, we do not want plug-ins to be kept apart, but to interact with each other. For this purpose, we adopt the notion of *hooking* or *extension points* which are associated with well-defined tasks that contributing plug-ins (*extensions*) can implement. Finally, we want a handy, easy to use, and, most importantly, lightweight framework, since applications are supposed to run client-side with as little overhead as possible.

The design of our framework and Spamato was accompanied by the development of a *rating system* which we call Trooth. For our work, we use it in two distinct areas: First, as a utility to comfort the usage of foreign plug-ins and, second, as a reputation management system in a collaborative spam filtering process. For the first application, we request users to rate a plug-in in terms of security ("does a plug-in really perform what it is supposed to do?") or usability ("does the usage of a plug-in provide any benefit?"). In the second domain, we separate spam from legitimate messages by requesting users to vote for them. Trooth has been designed to be robust as it withstands malicious users who try to deceive the system, partially decentralized as clients are explicitly involved in the voting and evaluation process, and collaborative and personalized as users interact with each other for collective benefits.

The remainder of this paper is organized as follows. In the next section, we discuss related work. In Section 3, we describe our plug-in framework. Section 4 introduces the Trooth rating system. We show in Section 5 how to embed the plug-in framework and the Trooth system in the Spamato spam filter system. And finally, we draw a conclusion in Section 6.

2 Related Work

In this section, we give an overview of related work, separated into recent plug-in frameworks, recommendation systems, and spam filter systems.

Plug-in Frameworks The Eclipse [1] framework primarily focused on integrated development environments (IDEs) and plug-ins which facilitate the building of applications. With version 3.0, the *Rich Client Plattform* (RCP) allows for the development of any client applications. The Eclipse Runtime, which reflects the minimum set of classes to build a rich client application, is built on top of the OSGi framework [2], that defines, for instance, the life cycle of a plug-in. The orientation on business compatible solutions and the emphasis on *rich* client applications clearly shows that Eclipse does not aim to provide a *lightweight* plug-in container but a powerful solution for all circumstances. A minimal "Hello World" project in Eclipse has a footprint of about 450 kB, while our plug-in framework creates about 50 kB only. Our approach adopts some of Eclipse capabilities. Mainly the notion of extensions and extension points as well as the declaration of plug-in interdependencies in a `plugin.xml` file is shared by us. But while Eclipse employs a service locator to connect plug-ins, we use the constructor injection pattern to automatically resolve dependencies among plug-ins.

Apache Tomcat [3] is a container that supports the Java Servlet and JavaServer Pages specifications, running J2EE web server applications. In that, it differs from our approach that rather aims to be embedded in client side applications. Using Tomcat, dependencies can be modelled by using globally shared directories, which is also reflected in a simple class loader hierarchy. In contrast to our approach, where each plug-in can share parts of its classes (using a `SharedClassLoader`) and use contributed classes from other plug-ins (using a `DependencyClassLoader`), in Tomcat, only one class loader exists that enables plug-ins to share classes among each other. Furthermore, Tomcat does not provide any means of extensions or extension points.

The Apache Avalon framework is implemented in several projects, such as Phoenix, Fortress, Merlin, or Codehaus' Loom, see [4] for a description of the project history. We compare our approach to Fortress [5], the only project which provides a lightweight plug-in container. In Fortress, plug-in dependencies are declared as “inline” JavaDoc tags directly in the source code. In contrast, we manage all plug-in dependencies and information in a separate file. Although the Fortress approach seems to lead to less overhead, our scheme has the advantage to encapsulate all dependencies in one single file. Furthermore, Fortress does not provide extensions and extension points and uses a service locator instead of the constructor injection as we do.

The Codehaus PicoContainer [6] offers a very simple container facility with a footprint of about 50 kB. It provides the constructor and setter injection patterns but has to be configured directly in the source code. No separate descriptor file is needed to model interdependencies among plug-ins since all plug-ins run with the same class loader.

A NanoContainer [7] bundles several PicoContainers and allows for the usage of separate class loaders. Additionally, dependencies can be declared in many scripting languages, while we stick to an XML description. The main differences to our approach are that NanoContainer, first, does not implement extensions and extension points, and second, cannot be altered after the startup—it does not provide install or update features as we use them in our plug-in container.

Recommendation Systems TROOTH has been influenced by work in the domain of recommendation systems, also known as collaborative filtering, see [8, 9, 10]. In these systems, the concept of predicting future user behavior or recommending suitable choices based on historical data is common. We share this idea but focus on specific practical aspects. For this purpose, we only provide a heuristic to evaluate an item for a user, we do not provide a mathematical analysis.

In contrast to *offline* algorithms, such as [11, 12, 8], that usually recommend a single item to a user based on a fixed set of votes, we consider a continuous stream of items which a user has to assess. These items can arrive at anytime, making predictions time-dependent. Furthermore, users cannot be asked to *train* a server-based system in order to increase the rate of correct predictions; items are always selected client-side. We also emphasize the role of clients in that we store only a minimum of data on the server and evaluate items on clients. We differ from other *online* approaches, such as [13, 14], in that we do not consider a round-based synchronous model. Again, in TROOTH, users can vote for items at any time.

Spam Filter Systems One of the most well known spam filtering tools is the Apache SpamAssassin [15]. SpamAssassin uses an extendable rule system to classify email messages. It is easy to extend SpamAssassin with custom rules that, for instance, employ Bayesian filtering [16], query URL blacklists [17], or exploit other filter systems such as Razor [18]. In contrast to SPAMATO, it is running server-side, has no email client add-on, and, thus, only rudimentarily supports a user feedback channel—to report a missed spam message a command line call needs to be executed.

SpamGuru [19] is another server side filtering system developed by IBM. Unlike SpamAssassin and SPAMATO, SpamGuru is a closed source project which cannot be extended by external developers. SpamGuru uses an optimized ordering of filter mechanisms to maximize the message throughput of a server. A plug-in for the Lotus Notes mail client provides a convenient user feedback channel that is used to report spam messages to a collaborative spam filter which is part of the SpamGuru system. Clients other than Lotus Notes are currently not supported.

The Cloudmark Desktop [20] is a commercial Microsoft Outlook and Outlook-Express add-on. It employs several filtering techniques to identify messages but mostly relies on collaborative filters. The Cloudmark Desktop is the commercial spin-off of Vipul’s Razor [18]. Both access the same central spam database which collects (manual) spam reports from all users of the system. Although Razor is an open-source project, the server-side components have not been published. Thus, it is not possible to extend the system to integrate additional, custom collaborative filtering techniques. Furthermore, their trust system “Truth Evaluation System” (TeS) is kept secret to prevent manipulation.

On the client side, a variety of email filters are available. Most of them use a Bayesian filtering algorithm and are bound to a specific email client, limiting their effectiveness and application domain. SpamPal [21] is—besides SPAMATO—the only client side filtering suite we know about which is designed to support several spam filters and to be email client independent. Instead of using a special email client add-on, SpamPal acts as a transparent proxy between the email client and the email server. SpamPal is an open source project and custom extensions have to be implemented in C. Unlike Spamato, SpamPal runs on Microsoft Windows only and does not provide a user feedback channel making it impossible to employ collaborative filters.

3 The Plug-in Framework

In this section, we describe our plug-in framework which has originally been built to facilitate the development of third-party filters for the Spamato spam filter framework (see Section 5), but has basically been designed to ease the implementation of any plug-in based application. However, some of our examples are still described in the context of Spamato for clearness without loss of generality.

When talking about plug-ins, on the one hand, we mean software components that are rather *independent blocks* of code. They usually do not provide any features, such as classes or resources, to other components, and do not make use of any shared features. Independent components bundle everything they need to perform a specific task and do not interact with any other components. On the other hand, we also think about *open framework blocks* which generally do not only provide features to other components but explicitly incorporates them, exploiting their capabilities to fulfill a job. It is obvious, that the latter is more powerful and thus includes the former component type.

In the next section, we formulate the characteristics of our plug-in framework and show how to meet the requirements to support the component types describe before. After that, we explain the general process when starting, how plug-ins are connected, loaded, and configured. Finally, we highlight the deployment mechanism to publish, install, and update plug-ins.

3.1 Plug-In Characteristics

A plug-in features some apparent characteristics such as a name, a description, and a main class. These parameters are generally necessary to manage plug-ins, provide information to users, or initialize them. Additional requirements include a security facility to restrict the access to local resources, a deployment mechanism to manage different versions of plug-ins, and a scheme to model dependencies between plug-ins.

3.1.1 The `plugin.xml` Descriptor File

Most of these characteristics are mapped to a `plugin.xml` file which describes a plug-in and its dependencies. Listing 1 exemplifies the `plugin.xml` descriptor of a dummy plug-in.

The aforementioned `<name>` and `<description>` of a plug-in, which are solely of descriptive usage, are listed in lines 2 and 3. The `<class>` denotes a plain old Java object, it does not have to inherit from a “PlugIn” class or implement any interface. In Section 3.3, we describe how plug-ins are initialized and loaded. The `<version>` and `<update-url>` tags provide information for the deployment mechanism which is detailed in Section 3.6.

The `<requires>` section of the XML file specifies security requirements and dependencies on other plug-ins. In this example, the Dummy PlugIn requests `"all"` permissions meaning that the plug-in must not be applied any restrictions enforced by a Java SecurityManager to which permissions are directly mapped. Therefore, this concept also allows for a fine granular assignment of permissions such as the read/write access to local files from a user directory or the connection to a specific web server only. This is particularly important when dealing with third-party, untrusted plug-ins as described later. Additionally, in the `<requires>` section, a plug-in defines its dependencies on other plug-ins—either to get access to `"<share>"`d classes or resources, or to subscribe to offered “extension points.” We describe extensions and extension points in Section 3.2.

The `<share>` part enables other plug-ins to extend or use the facilities provided by the sharing plug-in. In this example, the “Dummy PlugIn” allows other plug-ins to access all classes in the package `ch.ethz.dcg.dummy .shared` and additionally the `ImportantSharedClass`. The sharing of resources such as images or files can similarly be achieved. Line 16 states the publishing of an extension point which is further described in the following section.

Listing 1 A Dummy `plugin.xml` Descriptor File

```
 1  <plugin>
 2    <name>Dummy PlugIn</name>
 3    <description>This is a very simple dummy plug-in.</description>
 4    <class>ch.ethz.dcg.dummy.DummyPlugin</class>
 5    <version>1.0</version>
 6    <update-url>http://spamato.ethz.ch/update</update-url>
 7    <requires>
 8      <permission type="all"/>
 9      <plugin key="another_dummy"/>
10        <extension point="dummy_point" param="hello world" class="ch.ethz.dcg.dummy.DummyExtension"/>
11      </plugin>
12    </requires>
13    <share>
14      <package name="ch.ethz.dcg.dummy.shared"/>
15      <class name="ch.ethz.dcg.dummy.ImportantSharedClass"/>
16      <extension-point id="my_dummy_point"/>
17    </share>
18  </plugin>
```

3.2 Extensions and Extension Points

The concept to extend other plug-ins has been borrowed from Eclipse. Plug-ins offer well defined *extension points* which other plug-ins as *extensions* can register with. This concept resembles a publish/subscribe mechanism but is much more powerful: Registered extensions are not only notified to handle an event, but are expected to extend the capability of the extension point or to perform an expected job.

In Eclipse, for instance, many plug-ins add information or views to the user interface by hooking into extension points that are called when the GUI is shown. Thus, new visible elements with associated tasks are embedded into the default editor. The Spamato system offers several extension points (see Section 5), one for plug-ins which perform a spam filtering task. Whenever a new email arrives, it is checked by each registered extension which returns whether it is classified as spam or not. In this case, the filtering process is extended or rather relies on what registered extension contribute; the control flow runs through registered filters. In Section 3.6, we describe an extension point that enables the deployment mechanism to support additional update protocols.

In Listing 1, `<extension-point>`s are defined in the `<share>` section of a plug-in descriptor file (line 16). The `"id"` can be referenced in the `<requires>` part of another plug-in. Besides the `<class>` that implements the `<extension>`, additional static values can be assigned to parameterize the registration (line 10). Usually, the main class of an extension implements an interface which is shared by the plug-in that offers the extension point. In the Spamato example above, a plug-in that registers as a spam filter also implements the `SpamFilter` interface that belongs to the offering plug-in.

3.3 Dependency Modeling and Class Loading

Our plug-in mechanism basically represents a lightweight container component which loads plug-ins in a specific format from a specified directory. Plug-ins provide their class files in a `classes` directory, additional jar files in a `lib` directory, and static resource files, such as documentation files, in an `etc` directory. These directories are located below a `bin` directory which also holds the `plugin.xml` file. Further dynamic content which is created at runtime, such as user defined config files, are saved in the root directory of each plug-in.

All plug-ins are located in a directory which is recursively traversed when the container is started. As mentioned earlier, a `plugin.xml` file characterizes the interaction (required plug-ins for shared classes and extension points) with other plug-ins. These files are parsed in order to build a directed, acyclic graph which reflects all dependencies of all plug-ins. The graph is used, for example, to determine a start-up and a shutdown order, to have plug-ins available when dependent ones need them. Please notice that the container is partly implemented as a plug-in itself (the *Runtime* plug-in). It shares classes and resources, offers extension points, and exhibits all other features of a normal plug-in. Thus, the Runtime is also contained in the graph but does not depend on any other plug-in.

This graph is also reflected in a similar hierarchically organized set of Java `ClassLoaders`. Generally, each plug-in is managed by its dedicated class loader. By default, no plug-in can use or even knows about other plug-ins; they are totally shielded in their personal namespaces. This results in four nice features. First, developers do not have to worry about other plug-ins. They can label their packages without considering problems due to any name collisions even though all plug-ins are dynamically loaded into the same JVM. More precisely, developers can even prohibit the access to their classes. Second, we can easily assign different individual security permissions as mentioned earlier. Third, the testing and analysis of plug-ins with different parameters is facilitated. In the SPAMATO system for instance, spam filters, which are themselves plug-ins, can be used multiple times in one Spamato instance with different settings just by copying them into different directories in the plug-ins directory. Thus, it is possible to easily compare the results of the same filters running at the same time with different settings. And finally, using separate class loaders provides the capabilities to update plug-ins without the need for a restart of the whole container component. Instead, only the updating plug-in and its dependent plug-ins need to be restarted which can be performed at runtime. We further detail this point in Section 3.6.

On the other hand, though, we still have to provide the sharing of classes, resources, and extension points. This means, we need some facility to let other plug-ins make use of the shared information. Additionally, hooking into an extension point entails the plug-in which offers the extension point to call methods of the extending plug-ins, mutually connecting both plug-ins with each other.

To allow for such interactions, the default class loader scheme has to be adapted. In our plug-in system, each plug-in is backed by four different types of class loaders: the *FileClassLoader*, the *SharedClassLoader*, the *DependencyClassLoader*, and the *CombinedClassLoader*. The interaction of two plug-ins and their class loaders is depicted in Figure 1.

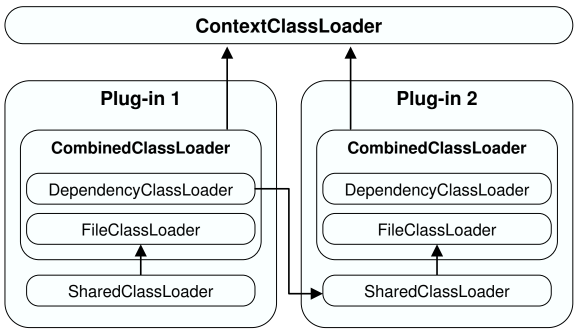


Figure 1: The class loader hierarchy of two plug-ins.

The *FileClassLoader* is responsible to load files from the file system, restricted to the associated plug-in directory. The *SharedClassLoader* wraps the *FileClassLoader*. It restricts the access for other plug-ins to those files which are declared as "shared" in the plug-in descriptor file. The *DependencyClassLoader* enables plug-ins to access shared classes of other plug-ins by using their *SharedClassLoader*, provided that a dependency is declared. And finally, the *CombinedClassLoader* combines the *FileClassLoader* and the *DependencyClassLoader* and provides all class files, resources, and libraries accessible through them to the associated plug-in. Furthermore, there is a single *ContextClassLoader*. It enables all *CombinedClassLoaders* to access the default Java classes, the bootstrap plug-in classes, which are not part of any plug-in, and all other classes and libraries that can be found in the default `CLASSPATH`.

Also note that our class loading mechanism differs from the default "first parent/then child"-scheme. We first browse the directly associated FileClassLoader, and only if the class or resource can not be accessed, the parent CombinedClassLoader is called.

3.4 The Life Cycle of a Plug-in

The life cycle of a plug-in in our framework is quite simple. It can be extended by implementing specific interfaces instead of hooking into an extension point.

The plug-in descriptor file is loaded and parsed in the initialization phase by a *Plugin Handler*; for each plug-in exists one handler. After all plug-ins have been initialized, they are loaded respecting the order of the directed acyclic graph described in Section 3.3. Thus, whenever a plug-in is loaded, its required plug-ins are available. The loading and instantiation of classes is performed using the *Dependency Injection* pattern (IoC) or more precisely, the constructor-based injection variant of it. Thus, references to required plug-ins are automatically assigned through constructor parameters which are resolved using Java Reflection.[1] The *start* and *dispose* phase can optionally be implemented by implementing the correspondent interfaces.

3.5 Configuration

An interfaces unifies the access to configuration settings from various sources. The settings of a plug-in can automatically be assigned as a constructor parameter by the plug-in framework during the start-up phase as described in the previous section.

Currently, text, Java properties, and XML files are supported; additional formats can be contributed using the correspondent extension point of the runtime plug-in. For instance, a database implementation would be more appropriate to support a large number of users for a server-side Spamato version.

3.6 The Deployment Mechanism

Oreizy et al. [22] identify three types of architectural changes in the life-time of plug-ins in a framework: the *addition* of plug-ins, the *removal* of plug-ins, and the *replacement* of plug-ins. We refer to these types as the *installation*, *deletion*, and *update* of plug-ins respectively. Furthermore, we extend our framework to allow for another aspect: the *publication* of new plug-ins by any user.

The *Runtime* plug-in provides four extension points to implement these four aspects in our framework; this is illustrated in Listing 2. Plug-ins can contribute to these extension points by implementing corresponding interfaces.

Listing 2 The Runtime plug-in offers extension points to provide deployment handlers and registers default ones.

```
<plugin>
  <name>Runtime</name>
  ...
  <share>
    ...
    <extension-point id="deploy.search"/>
    <extension-point id="deploy.download"/>
    <extension-point id="deploy.upload"/>
    <extension-point id="deploy.publish"/>
  </share>
  <extension point="deploy.search"
    id="http" handler="HttpSearchHandler"/>
  <extension point="deploy.download"
    id="http,ftp,file" handler="..."/>
  ...
</plugin>
```

Search handlers are used to find plug-ins that can be installed or updated. We provide default search handlers for HTTP and FTP servers as well as for the local file system (see Section 3.6.1). The search provides a list which contains fragments of the `plugin.xml` of a plug-in: informative data, such as the name and the description, as well as important data, such as the version number and the download mechanism. *Download* handlers fetch plug-ins from a download server. As an additional plug-in, the tracker-based Peerato system allows to download directly from other users [23]. *Upload* handlers store plug-ins on a download server, and *publish* handlers update the list of available plug-ins accessed by search handlers.

Please notice that the *Runtime* plug-in only provides the basic capability to manage the deployment cycle of plug-ins. But to employ it in a *user friendly* way in an application, further steps have to be taken. In the Spamato system, we use a browser-based approach to cope with this issue.

3.6.1 The Profile Deployment Scheme

On multi-user platforms, such as Linux and Windows XP, it is often feasible to install an application based on our framework for all users only once. This eases the application maintenance, for instance, when updating plug-ins. Still, users should

[1] We also provide the service locator approach by allowing plug-ins to access the plug-in container. But we regard the constructor injection method to be easier to maintain—and a reference to the service locator can only be accessed in this way.

be able to configure their environment, for instance, by installing custom plug-ins or removing default ones.

We address this issue in our *profile deployment scheme*. Usually, an administrator installs an application to a directory which users can only read from, the *default* installation directory. To allow for the configuration of a user-specific environment, plug-ins are installed to the user's *profile* directory the first time a user starts an application. Any modification can now be performed in the profile directory instead of the read-only installation directory. Nevertheless, administrators can update the default installation and our profile deployment scheme applies the changes to each user profile.

4 The Trooth System

In this section, we introduce TROOTH as a robust, partially decentralized, collaborative, and personalized rating system. Most examples will be given in the context of spam filtering as we apply it in the SPAMATO spam filter system (see Section 5). But, generally, it has been designed to be independent of SPAMATO and, for instance, has been used to comfort users to employ foreign plug-ins as well.

4.1 Preliminaries

In this section, we describe some basic aspects which we use and refine in the next sections.

The general aim of our rating system is to express the evaluation of items, such as people, plug-ins, or emails, to be either *good*, *bad*, or *unknown* by allowing users to classify the item (in some context) as *good* or *bad*.[2] In a nutshell, given an item, a user first tries to revert to a recommendation of the rating system to check whether it is *good* or *bad*. If the *evaluation* calculated by the rating system has been *unknown*, she has to manually assess the item. Afterwards, she casts a *vote* to express her personal assessment.

In this section, we assume that a pre-defined, globally valid evaluation for an item exists which has to be exposed for each item. In Section 4.2, we revise this assumption and instead calculate *individual opinions* about each item for each user.

[2]We assume that a user who does not know how to classify an item does not vote at all.

4.1.1 Evaluation Functions

A global evaluation of an item can be derived from all *user votes* in various ways. For instance, using a simple *majority* evaluation, the overall categorization of an item is *good* if a majority of all users ($> 50\%$) votes in favor of the item, *bad* if a majority votes against the item, and *unknown* if the number of *good* and *bad* votes are equal. For the more general *threshold* evaluation function, we classify an item to be *good* (*bad*) if the fraction of good (*bad*) votes of all votes is larger than a threshold value h_g (h_b) and *unknown* otherwise.

It is easy to extend this *simple* voting scheme from the set $\{good, bad, unknown\}$ to a range where votes and evaluations can be in the interval $[0, 1]$. In this case, we can define the result of an evaluation to be the average of all vote values.

4.1.2 Weighting Votes With Trust Values

So far, votes or rather users have been considered to be equally important. In real life, however, it can be beneficial to apply different weights to votes or, in other words, to consider some users to be "more equal than others." Since we assume the existence of a single, pre-defined evaluation for each item, users who often agree with the majority of user should be trusted more than those who regularly dissent.

Ideally, this means trying to separate users into two groups: One group contains those users who are *trustworthy* and the other group those who are *malicious*. Practically, it is possible to approximate these groups by introducing *trust values* for each user that are adjusted whenever new information is available. Then, instead of simply summing up equal *good* (and *bad*) values as before, each vote is previously weighted with the trust value of the associated user. For this approach to make sense, we generally assume that the group of trustworthy users are a majority, or more precisely: that those users who agree with the majority are trustworthy.

The *Additive Increase, Multiple Decrease* (AIMD) approach takes user specific and automatically adjusted trust values into account. When all users have cast their votes, the trust values are modified. Using AIMD, users who voted *correctly*, that means in accordance with the majority, are awarded by slightly increasing their trust values.

On the other hand, users whose votes do not comply with the majority are punished by harshly decreasing their trust values.

Please note that we are rating now in two different domains: On the one hand, we want to evaluate items by having unknown users vote *good* or *bad* for them. On the other hand, we want to calculate trust values for unknown users to make votes more reliable. The voting (and thus also the evaluation) is actively performed; trust values are implicitly generated. While in principle it is possible to let users choose whom they want to trust, in reality this is considered too involved.

4.1.3 Implementation Issues

For applications like the collaborative spam filter system Spamato, the voting for an item (in this case, a message) and the categorization of it (spam/not spam) will not take place at a single point in time. Instead, users can always vote for an arbitrary message, and Spamato classifies a message whenever it arrives in a user's inbox. Additionally, not all users vote for all items, since not all users receive the same messages.

In the context of plug-in assessment, a user is free to select a plug-in whenever she likes and wants to know whether it is a useful plug-in from a trustworthy developer (*good*) or not (*bad*). And clearly, not all users will necessarily select all plug-ins since user interests are not the same.

Therefore, user votes have to be stored for later usage and the evaluation of an item has to be recalculated whenever a new vote has been cast. In other words, evaluations are time-dependent. Furthermore, users must not be able to vote more than once for the same item or a scheme must exist to handle multiple votes in a reasonable way. Finally, users casting a vote have to be authenticated to prevent manipulation.

Implementing the AIMD approach or weighting algorithms in general entails some difficulties. As users can vote at any time, the update of trust values either has to be conducted at a specific point in time, or needs to consider earlier changes associated with the same item. While an algorithm for the former solution can calculate the new values by knowing few variables only, such as the time of the first vote, the current time, and the number of votes so far, it obviously ignores later votes and thus important information to provide fair trust values. On the other hand, adjusting the trust values only once reduces the complexity of the system and saves server resources. The latter solution, however, means that we need to manage extensive historic information about the voting process. Additionally, trust values have to be updated every time a new vote is called, thus, increasing the demand for server resources. In both cases, the server has to store the trust values for each user and the overall evaluation of each item—to avoid instant time- and resource-consuming calculations whenever these values become necessary.

4.2 Motivation for Trooth

In the previous section, we have assumed that it is possible to globally evaluate an item—that an overall evaluation exists which coincides with the votes of all trustworthy users. But the separation of users into groups of trustworthy and malicious users, as described in Section 4.1.2, often is too harsh. In fact, the assumption that an objective overall categorization can always be calculated is the weakest point in the ideas described so far.

Instead, we believe it is more reasonable to individually evaluate an item for each user separately. A user does not distinguish between trustworthy and malicious users anymore, but between users who generally vote in accordance and those who do not. Thus globally seen, users are implicitly separated into several *special interest groups* (SIGs) who share a "similar opinion" rather than to discriminate them with the "black & white" scheme described before.

Please notice that we still believe that trustworthy and malicious users exist. While the former describe users who really try to express their opinion, the latter usually vote against the common sense and try to deceive the system, probably for personal benefits. We also consider users who make failures and others who just do not understand how to operate a voting system. So generally, from a user's point of view, all these categories can be reduced to assenting and dissenting users only. For simplicity, in this section we use the term *malicious* for byzantine as well as incautious and unaware users.

Depending on the voting domain, the number of groups might vary significantly between only a few and tens or more. Although we expect groups to be rather large and overlapping, in the extreme, each user might trust only herself so that the number of groups equals the number of users. But this especially expresses the strength of our system: Even if all except one user are malicious, this one will (eventually) figure out not to trust anybody except herself. Thus, the system can even serve different minorities with satisfying results while approaches that assume the existence of an objective evaluation cannot.

Since TROOTH does not compute a global evaluation of an item for all users, it is possible to reduce the consumption of server-side resources to a minimum. Therefore, in TROOTH we store only (item,user,vote)-tuples server-side and calculate user specific trust values client-side as we show in the next two sections. In Section 4.5, we sketch how to extend TROOTH in several directions, including a complete peer-to-peer solution.

4.3 Managing Votes and Trust

As in structured peer-to-peer systems, we assign each item and each user a unique identifier from an interval $[0, ..., N]$, organized as a "ring." Thus, we can use the notions of *clockwise* and *anti-clockwise* to denote neighbors on the ring. In Section 5.3, we show how user IDs (and signatures to authenticate users) are generated in SPAMATO.

4.3.1 The Voting Process

When a user votes for an item, she sends her opinion (*good* or *bad*) to the TROOTH server and locally adapts the trust values for other users who voted for the same item. In more detail, the voting process takes the following steps:

- User u_0 sends a vote v_0 for an item i to the server where the (i, u_0, v_0)-tuple is stored.
- The server assembles two lists which are populated with the identifiers of other users who voted *good* (list G) and *bad* (list B) for item i before. Each list contains a maximum of k user IDs that are numerically nearest to u_0 in respect to the ring formation. The lists G and B are sent to the client.
- User u_0 locally adapts the trust values of the users sent to her by increasing the trust values of those users who agreed with her own vote v_0 and decreasing the trust values of those users who voted against it (using the AIMD approach described in Section 4.1.2 or any other weighting scheme).

The user can adjust this process by sending the value for k to the server.

4.3.2 The Evaluation Process

To classify an item, *good* and *bad* votes from the server are weighted with the client-side stored trust values. In detail, the following steps are performed for the evaluation process:

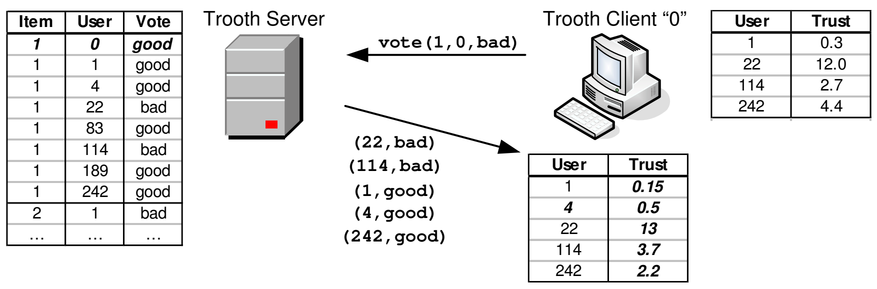
- User u_0 sends a query for item i to the server.
- The server returns two lists containing identifiers of users who voted *good* (list G) and *bad* (list B) as in the second step of the voting process described above.
- User u_0 extracts the $l \leq k$ most trustworthy users of each list, resulting in the lists $G' \subseteq G$ and $B' \subseteq B$.
- Finally, the classification can be calculated using the threshold evaluation function described in Section 4.1.2 with $V' = G' \cup B'$ (or rather all weighted votes of the selected users).

Again, the user can adjust this process by sending the values for k and l to the server.

4.3.3 Discussion

In the second step of both algorithms, the server returns about $k/2$ clockwise and anti-clockwise neighbors of user u_0 for each list. If there are less then k other users who voted *good* (*bad*) for item i, we return only that many without any loss in quality. Assuming that users usually vote for the same *type* of items[3], it is reasonable to believe that the total number of trust values that have to be handled client-side is bounded. This means that each

[3]Regarding e-mails, for instance, users who are collected on the same e-mail address list often get the same spam messages. Regarding plug-ins or products, Amazon-like "Users who bought this product also bought..." statements also underline our assumption that users will often vote for the same type of items.



Item	User	Vote
1	***0***	***good***
1	1	good
1	4	good
1	22	bad
1	83	good
1	114	bad
1	189	good
1	242	good
2	1	bad
…	…	…

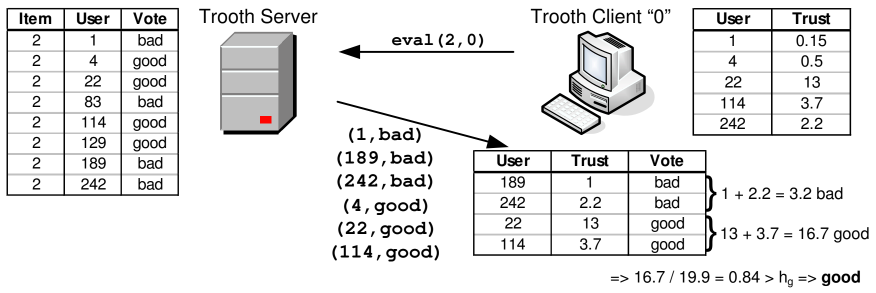
```
vote(1,0,bad)
```

```
(22,bad)
(114,bad)
(1,good)
(4,good)
(242,good)
```

User	Trust
1	0.3
22	12.0
114	2.7
242	4.4

User	Trust
1	***0.15***
4	***0.5***
22	***13***
114	***3.7***
242	***2.2***

Figure 2: This figure exemplifies the voting process. User “0” casts a *bad* vote for item “1” and adjusts her trust values.



Item	User	Vote
2	1	bad
2	4	good
2	22	good
2	83	bad
2	114	good
2	129	good
2	189	bad
2	242	bad

```
eval(2,0)
```

```
(1,bad)
(189,bad)
(242,bad)
(4,good)
(22,good)
(114,good)
```

User	Trust
1	0.15
4	0.5
22	13
114	3.7
242	2.2

User	Trust	Vote
189	1	bad
242	2.2	bad
22	13	good
114	3.7	good

1 + 2.2 = 3.2 bad

13 + 3.7 = 16.7 good

=> 16.7 / 19.9 = 0.84 > h_g => **good**

Figure 3: This figure exemplifies the evaluation process. User “0” requests the votes for item “2” and evaluates it as *good*.

user generally stores only a small subset of all users who share the same opinion. It is also an advantage that a user's vote can only affect those users in the implicit neighborhood. Thus, the impact of a possibly malicious user trying to cheat the system is limited. On the other hand, though, the rather high consumption of bandwidth for each voting and evaluation operation can be regarded as a drawback.

By choosing only the most trustworthy users in the third step of the evaluation process, we decrease the influence of unwanted users to a minimum. As an additional optional step, trust values can be adjusted after the evaluation process similar to the last step in the voting process. Doing so would amplify the influence of trust values even more. Furthermore, the calculated evaluations should automatically be sent as a personal vote to the server. If the user does not agree with the evaluation, she would (immediately or after a short while) send her correct opinion rejecting the old one.

Please note that a malicious user who tries to gain a high trust value in order to manipulate the evaluation process, previously would have to "play by the rules" for a long time and, thus, helping other users more than harming the system. Furthermore, to manipulate a particular user (or a group of users with assenting opinions), it is necessary to, first, get an ID that is near to that of the user, and second, to know which items the user is "interested" in. While attacking one particular user will be hard, it is almost impossible to oppose against many groups or even all users at once. Thus, TROOTH significantly reduces the impact of malicious users in the evaluation process.

4.3.4 Example

Figure 2 exemplifies the voting process ($k = 3$, AIMD will add 1 to the trust value of assenting users and multiply the trust value of dissenting users by 0.5, and the user identifier space is in the range of 0 to 255). First, client "0" sends her vote "bad" for item "1" to the TROOTH server. The server inserts the vote into the voting table and populates two lists for good and bad votes of users whose identifiers are numerically nearest to 0. Since only 2 users have voted *bad*, this list is returned with these two entries only (22 and 114), while the list of good votes contains three entries (1, 4, and 242). Next, user 0 adjusts the trust values using the AIMD approach. Since she voted *bad*, users 22 and 114 are awarded, and users 1, 4, and 242 are punished by increasing or decreasing their trust values respectively. For instance, user 22 has an old trust value of 12.0 which results in a value of 13.0 after increasing it by 1. Similarly, user 242 has an old trust value of 4.4 which is decreased to 2.2 after multiplying it with 0.5. User 4 has not been in the trust table before, therefore, she is rated with a default value of 1 before being punished.

In Figure 3, an example of the evaluation process is depicted ($k = 3$, $l = 2$, and the threshold for a *good* decision is $h_g = \frac{2}{3}$). After sending an evaluation request for item "2" to the server, user "0" receives two lists as described before. The client extracts $l = 2$ votes of each list which have been cast by users she trusts most (189 and 242 for bad, and 22 and 114 for good). Again, one user (189) has been unknown and was therefore rated with the default value 1, which was chosen since this value is higher than the third *bad* choice (user 1 with a trust value of 0.15). The trust values are accumulated and the evaluation is performed using the "threshold" evaluation method. Since the good votes are more trusted (16.7 to 3.2), the overall classification is *good*.

4.4 The Majority Heuristic

We introduce the *majority heuristic* which can be applied as a special case when many users have almost unanimously decided about an item. To use it, one should have ruled out the chance of malicious users being a majority.

In the *voting process*, the server still stores the vote of a user for an item. But the server does not return any data and the client, therefore, cannot adjust any trust values.

In the *evaluation process*, the server sends only the number of *good* and *bad* votes to the client. Therefore, the client is not able to select her most trusted users anymore; all votes count the same. The evaluation for the item is calculated using the majority or threshold evaluation function.

The user can completely configure the processes: The total number of voters and the fraction of dissenting voters have to be sent to the server; the parameters for the threshold evaluation function can be adjusted in the client.

The majority heuristic clearly simplifies the voting and evaluation processes by sending less information between client and server and, thus, also saving bandwidth. By doing so, it reduces the reliability of the classification since trust values are not considered anymore. But this can be neglected since there are almost none dissenting votes, and malicious users cannot be a majority.

4.5 Extending Trooth

In this section, we provide two extensions of the Trooth system aiming in orthogonal directions. While the first one describes a system which centers all activities at one server, the second sketches the idea of a completely decentralized approach.

4.5.1 Server-Side Trust Values

In the Trooth system described so far, each user stores a list of trust values of other users on the client. As we previously explained, although it is possible that this list contains a trust value for all other users, it is more likely to hold only a small subset of neighbors.

In contrast to our earlier motivation, we could store and adjust trust values and determine the classification of an item on the server. By sending (or storing) the parameters of the voting and evaluation processes from the client to the server, the user would still be able to adjust the outcome as before. Therefore, running Trooth server-side is no problem.

Having trust values stored on a single server also allows for a *global* view on this data. Consolidating the trust values of each user could disclose a variety of interesting information—for instance, how groups of assenting users look like or whether malicious users or rather users that nobody trusts exist. Another advantage is that users can now share their trust values between different machines or accounts. Furthermore, aggregated trust values could be used to provide a new user with some initial data. On the other hand, processing Trooth on a server drastically increases the resource demand for CPU and storage (while the bandwidth consumption is lowered).

We want to emphasize that a server-side Trooth system is not similar to approaches summarized in Section 4.1. The main difference is that we manage trust values for each user separately, still assuming that it is more promising to rely on several groups of assenting users than on trustworthy and malicious (in its original sense) ones.

4.5.2 Distributed Trooth

Trooth shifts most of the work to the client, keeping only the storage of (item,user,vote)-tuples on the server. This is a good basis to completely decentralize the Trooth system.

We propose the usage of a distributed hash table (DHT) such as Chord [24] or Kademlia [25] to obtain a server free Trooth system. In such systems, the “lookup” operation is the most important command which maps a *key* to the peer being responsible for it. Besides this, the “store” operation stores the *value* associated with a key at the managing peer.

Regarding Trooth, user votes have to be managed in the DHT. Therefore in the voting process, (item,user, vote)-tuples are the values to be stored at the peer responsible for the item. For this approach to work, the mentioned DHTs have to be adapted only slightly to support the storage of multiple values for one key. Similarly in the evaluation process, a user performs a lookup for an item ID and the responsible peer has to return a subset of all votes that have previously been cast. Thus, a fully decentralized, peer-to-peer styled Trooth system can generally be realized.

There are, however, some difficulties. The voting and evaluation processes will take more time due to the nature of a DHT where the responsible peer has to be looked up by routing to several intermediate stations. Furthermore, a DHT has to manage other issues such as the handling of joining and leaving peers, counter measurements for hot spots, and caching and replication mechanisms. While trust among peers can also be regarded as a key task of DHTs, we want to describe this problem shortly in relation to Trooth.

In Trooth, we assume the existence of unique user identifiers as well as the possibility to verify which user has cast which vote. In Subsection 5.3, we describe how we guarantee these assumptions in the Spamato system by generating a public/private-key pair for each user which is used to sign votes. Although this approach is server-based, we could still employ it for generation and verification of keys only. For a pure distributed approach, though, it is necessary to abandon this server, too.

Another drawback is that peers cannot be trusted. A peer is able to alter information in any way before sending it to a user. Thus, votes can be modified, coined, or deleted at will. While the signing of votes will help to detect modified or coined votes, peers cannot be kept from concealing them. One solution to this problem is to store votes not only at one peer but at many. Similarly, a lookup would have to return votes from several different peers. Although this increases the amount of data in the system and the effort to store and lookup it by the replication factor, the reliability of the result will be increased accordingly.

5 Spamato

In this section, we present the spam filter system Spamato which has initially been introduced in [26] with a focus on spam filtering strategies. Here, we highlight the systems aspects of Spamato, in particular, how it utilizes the plug-in framework and Trooth.

First, we give a rough overview of Spamato in Section 5.1. In Section 5.2, we exemplarily show how the plug-in framework is employed. After that, we describe the *Spamato Authentication and Authorization System* in Section 5.3 which we use to generate unique user identifiers for Trooth. Next, we show in Section 5.4 how Trooth is integrated into our collaborative spam filter to distinguish between spam and legitimate messages. And finally, we sketch in Section 5.5 how the plug-in mechanism employs Trooth to classify plug-ins.

5.1 The Spamato Spam Filter System

Spamato is a flexible, client-side spam filter system. As an add-on, it can be embedded in common email clients such as Outlook or Thunderbird, or can run independently as an email proxy. Emails arriving in a user's inbox are automatically checked by several spam filters, and detected spam messages (evaluated as *bad*) are moved to a special folder. The user can interact with Spamato by manually reporting messages which have not been identified as spam and revoke messages which have falsely been identified as such. This way, a user collaborates with the system, sending feedback (votes) to filters and the Trooth system.

Spamato has been in use for almost 18 months now. It is available for download at: `http://www.spamato.net`.

5.2 Using the Plug-In Framework in Spamato

Spamato consists of several plug-ins which implement different aspects of a collaborative spam filter system. The obvious key functionality of a spam filter system is to check whether incoming messages are spam. This task is performed in the *Spamato Filtering Process* (SFP) and is implemented by the *Spamato Base* plug-in. The SFP is depicted in Figure 4.

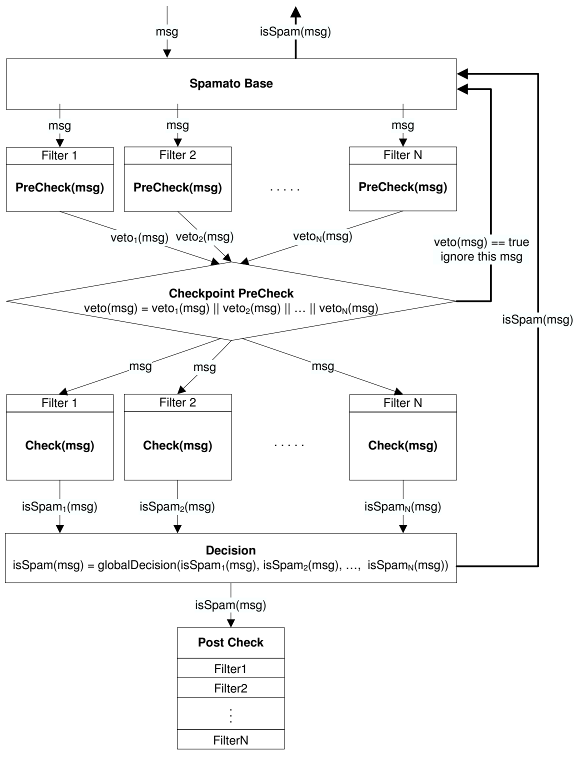


Figure 4: The filtering process consists of five phases which are mapped to extension points.

The SFP is separated into five phases. Generally, the procedure is triggered if a new message arrives. Then, each registered plug-in has the chance to *pre-check* the message in order to denote if the message has to be filtered at all. Subsequently, the real *checks* are performed and their results are accumulated to calculate the overall spam probability. Finally, the *decision* is returned to the user, and plug-ins can adapt to the decision in the *post-check* stage.

These phases are mapped to extension points which *PreCheckers*, *Filters*, *DecisionMakers*, and *PostCheckers* can contribute to. This is illustrated in Listing 3.

Listing 3 The Spamato Base offers several extension points to control the Spamato Filtering Process

```
<plugin>
  <name>Spamato Base</name>
  ...
  <share>
    ...
    <extension-point id="sfp.filters"/>
    <extension-point id="sfp.precheckers"/>
    <extension-point id="sfp.postcheckers"/>
    <extension-point id="sfp.decision_makers"/>
  </share>
</plugin>
```

Several other plug-ins exist and collaborate with each other. For instance, the *Web Config* plug-in is a local web server which offers users and plug-ins an easy-to-use configuration mechanism using a common web browser. The *Activity Manager* plug-in collects information about the SFP and offers emails and results to filters and statistic tools. The complete list and their interactions can be found on our homepage.

5.3 The Spamato Authentication and Authorization System

The *Spamato Authentication and Authorization System* (SAAS) is used to create unique identifiers for users who want to interact with the Trooth system. Additionally, SAAS generates a public/private key pair with which users (automatically) sign their votes to prevent any cheating and manipulation attempts.

Since Spamato (and therefore SAAS) is embedded into an email client, SAAS can make use of the authentication process between the email client and the server. In other words, if a user is able to receive an email that has been sent to him via SAAS, the user is "authenticated" also for Trooth.

In more detail, the first time Spamato is started, the SAAS client locally generates a public/private key pair. The public part of this pair and the user's email address is sent to an SAAS server (using a TCP connection) which in turn sends a random *challenge email* to the stated address. On receiving this challenge email, the client signs the message with its private key and sends it back to the server (again using a TCP connection). After that, the user is fully registered with the SAAS server which stores the user's public key and the (hashed) email address.

Please notice that the actual implementation is slightly more complicated to allow for a reregistration of users who want to use the same SAAS user account, for instance, on several machines. Additionally, the Trooth and SAAS servers need to exchange data so that the Trooth server can validate a user's signature. See [27] for a detailed description.

5.4 Collaborative Spam Filtering

The *Earl Grey* filter is a collaborative URL filter. When receiving a new message, it collects all URLs in the message, extracts the domains, and calculates a hash value of them. This hash value is sent to the Earl Grey server which queries a database to find out whether the message is spam or legitimate. Entries in the database are collaboratively inserted by users who report "spam" or revoke "legitimate" emails (or rather the calculated hashes). Thus, users help each other to filter spam messages.

Since not all users define the term "spam" equally—some also declare unwanted newsletters to be spam while others like to read about online drug stores—clearly, a system like Trooth is necessary to handle these different opinions.

In the context of Trooth, the hash values are the identifiers for items, users are identified with their email addresses (or their SAAS public key), and reports and revokes correspond to *bad* and *good* votes. As said before, to prevent malicious users from harming the system, votes are signed with a user specific private key. Additionally, the Earl

Grey server ignores multiple reports/revokes and removes contrary votes for the same message and user.

5.5 Plug-In Evaluation

A dynamic plug-in system allows for the extension of SPAMATO even during runtime. All filters, statistics, sound, logging, and several other components are connected as plug-ins to the SPAMATO core. Currently, the system consists of "trustworthy" plug-ins only, which are plug-ins that are bundled with the installation version of SPAMATO and are written by the same authors. The *Peerato* plug-in facilitates the publication of "third-party" plug-ins, such as custom filters [23]. These plug-ins can be provided by any foreign developer neither associated nor known to the authors of SPAMATO, so that plug-ins from these sources have generally to be suspected first.

Using TROOTH, filter developers (or plug-ins in general) can gain a good reputation by having users vote in favor of them. The primary goal is to evaluate a plug-in in terms of usability or security, that is, whether it harms (*bad* votes) the system in any way or not (*good* votes). If the usability should be rated, votes in the range of $[0, 1]$ are of more use.

The email address of a developer (or the hash value of a plug-in) as identifier on the one hand and the SAAS identifiers for a user on the other hand merge the plug-in mechanism and TROOTH to a powerful system to evaluate third-party contributions.

6 Conclusions

In this paper, we have presented a lightweight but powerful plug-in container. It is incorporated in the SPAMATO spam filter system and significantly helps to manage a well-organized software architecture. Furthermore, we employ the TROOTH rating system to assess plug-ins as well as emails.

SPAMATO has been in use for almost 18 months now. It is available for download at: `http://www.spamato.net`, the source code is published on SourceForge: `http://sf.net/projects/spamato`.

References

[1] "Eclipse Foundation," http://www.eclipse.org.

[2] "OSGi Alliance," http://www.osgi.org.

[3] "Apache Tomcat," http://tomcat.apache.org.

[4] "The Story of the Avalon Containers," http://wiki.apache.org/avalon/ContainerStory.

[5] "Apache Excalibur Fortress," http://excalibur.apache.org/fortress.

[6] "PicoContainer," http://picocontainer.codehaus.org.

[7] "NanoContainer," http://nanocontainer.codehaus.org.

[8] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins, "Recommendation systems: a probabilistic analysis," in *Proceedings of 39th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998.

[9] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl, "An Algorithmic Framework for Performing Collaborative Filtering," in *Proceedings of the 1999 Conference of the American Association of Artificial Intelligence (AAAI)*, 1999.

[10] B. Awerbuch, Y. Azar, Z. Lotker, B. Patt-Shamir, and M. R. Tuttle, "Collaborate With Strangers To Find Own Preferences," in *Proceedings of 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2005.

[11] J. Kleinberg and M. Sandler, "Convergent Algorithms for Collaborative Filtering," in *Proceedings of ACM Conference on Electronic Commerce (EC)*, 2003.

[12] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Analysis of Recommendation Algorithms for E-Commerce," in *Proceedings of ACM Conference on Electronic Commerce (EC)*, 2000.

[13] P. Drineas, I. Kerenidis, and P. Raghavan, "Competitive Recommendation Systems," in *Proceedings of 34th ACM Symposium on Theory of Computing (STOC)*, 2002.

[14] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Tuttle, "Improved Recommendation Systems," in *Proceedings of 16th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005.

[15] "SpamAssassin," http://spamassassin.apache.org.

[16] Paul Graham, "A Plan for Spam," www.paulgraham.com/spam.html.

[17] "SURBL - Spam URI Realtime Blocklists," www.surbl.org.

[18] "Vipul's Razor," http://razor.sourceforge.net.

[19] R. Segal, J. Crawford, J. Kephart, and B. Leiba, "SpamGuru: An Enterprise Anti-Spam Filtering System," in *Proceedings of the First Conference on E-mail and Anti-Spam*, 2004.

[20] "Cloudmark Desktop," www.cloudmark.com.

[21] "SpamPal," www.spampal.org.

[22] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based Runtime Software Evolution," in *Proceedings of 20th International Conference on Software Engineering (ICSE)*, 1998.

[23] M. Ackermann, "Spamato Goes P2P," Master Thesis, Federal Institute of Technology Zurich (ETHZ), 2005.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001.

[25] P. Maymounkov and D. Mazieres, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric," in *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[26] K. Albrecht, N. Burri, and R. Wattenhofer, "Spamato – An Extendable Spam Filter System," in *Proceedings of 2nd Conference on Email and Anti-Spam (CEAS)*, 2005.

[27] S. Schlachter, "Spamato Reloaded," Master Thesis, Federal Institute of Technology Zurich (ETHZ), 2004.

[28] R. Meier, "Spamato Plug-in Architecture," Semester Thesis, Federal Institute of Technology Zurich (ETHZ), 2005.