

# Development and implementation of AntispamLab

—

Creating a PlanetLab application for realistic  
evaluation of email filters

Master Thesis

Luis Hernández Hernández

Supervised by: Slavisa Sarafijanovic and Jean-Yves Le Boudec



ÉCOLE POLYTECHNIQUE  
FÉDÉRALE DE LAUSANNE

Computer Communications and Applications Laboratory, June 2007

# CONTENTS

<b>1. INTRODUCTION – MOTIVATION AND FEATURES.....</b>	<b>2</b>
<b>2. CHARACTERISTICS .....</b>	<b>3</b>
2.1. CREATED NETWORK .....	3
2.2. SUPPORTED SERVER-FILTER ARCHITECTURES .....	3
2.2.1. Piped filters .....	4
2.2.2. SMTP filters .....	4
2.3. STANDARDIZED FILTER INSTALLATION .....	5
2.4. SIMULATING CLIENTS AND USERS .....	5
2.4.1. User event schedule and time scaling .....	5
2.4.2. Time overlaps .....	6
2.4.3. Collecting data .....	6
2.5. DANGER SIGNALS .....	6
2.6. TESTING PROCESS.....	7
2.6.1. User reports .....	8
2.6.2. Filter reports.....	8
<b>3. EXECUTION TIMELINE .....</b>	<b>9</b>
3.1. AVAILABILITY CHECKING .....	9
3.2. DEPLOYMENT .....	9
3.3. TESTING .....	10
<b>4. DEVELOPMENT CHALLENGES.....</b>	<b>11</b>
<b>5. CONCLUSIONS .....</b>	<b>13</b>
<b>6. REFERENCES .....</b>	<b>15</b>
<b>7. APPENDIXES.....</b>	<b>16</b>
7.1. THE SCRIPTS .....	16
7.1.1. Starting the tool .....	16
7.1.2. Deployment .....	16
7.1.2.1. Availability checking.....	16
7.1.2.2. Distributing and installing software .....	17
7.1.2.3. The server installation script .....	18
7.1.2.4. Post-deployment.....	19
7.1.2.5. Re-deploying.....	19
7.1.3. TESTING PROCESS .....	19
7.1.3.1. Network reset and run launch .....	20
7.1.3.2. Log collection and results .....	20

# 1. INTRODUCTION – MOTIVATION AND FEATURES

Current spam filter testing systems require that the software under test is isolated from its normal network of operation. The absence of this environment prevents the filter from collecting information that would be otherwise available, such as user activity, mail bulkiness or cooperative relationships. State-of-the-art filtering software focus in exploiting this kind of information, rendering the tests performed under isolation only partially representative of a filter's capabilities.

For a more realistic alternative, a testing system featuring a simulated network of email clients, servers and users needed to be developed. Hence the motivation for this project - a tool that creates said environment, using widespread email software, and performs tests under a given time scale. Thanks to this functionality, data on how the filter reacts to user feedback, cooperative interactivity, or other network-related parameters can be rapidly collected.

The starting point for developing such a system was a preliminary tool [1], which was able to perform one testing run inside a set of virtual machines created within a single physical machine. Its code was written in Python and implemented by a former student, and apart from the basic functionality, its main limitation was that the number of network nodes was severely restricted due to relying on the resources of the physical machine.

In order to provide a testbed able to support such a realistic network, the access to a considerable amount of machines is an expected requirement. This constraint was overcome through PlanetLab [2], a consortium formed by more than 350 institutions, including many prestigious universities. It hosts nodes throughout the entire globe, offering free access to the researchers from its affiliated universities. The testbed consists of per-user, separated virtual machines within each node, so that each user has virtually exclusive access to the full network.

Nevertheless, the random nature of planetary-scale systems requires special attention to the adaptability of the developed code. The possibility of variations in the status or reachability of a network node demands far more robustness than would be necessary in a fully-controlled environment. Thus, this project devotes considerable effort in performing status checks and collecting available feedback from the machines involved in the most critical processes.

Equally important is the need for versatility, as the filters to test are likely to need completely different ways to install and operate. A standardized procedure is provided, which accepts the main types of filters and allows their automatic installation for later testing.

Finally, this projects features email client simulators that are configured as spammers or regular users, which will read and send email depending on a randomly generated sequence with configurable parameters. These simulators can interact with most SMTP servers and they provide a base from which to develop more complex models, allowing to further approach the behavior of the real entities they simulate.

The project was the topic of a conference paper that was accepted for The Fourth Conference on Email and Anti-Spam [3].

## 2. CHARACTERISTICS

AntispamLab (also called “the tool” hereafter) has been designed for use within PlanetLab. Alternatively, it can also work inside any network that replicates PlanetLab's characteristics.

The tool will run in a “master” machine, from which it will launch and control a number of remote processes in the rest of available machines from the network. After an initial deployment phase has been completed, the tool will be able to test the performance of email filtering software under a simulation of its usual working environment. The test will consist in a certain number of consecutive “runs”, or configurable-length periods of simulated email activity.

To perform these tasks, the only needed inputs are a text file containing the hostnames of the usable machines, a collection (corpus) of differentiated spam and non-spam (ham) mails, and the installation files corresponding to the filter to be tested.

The tool's features are further explained in this section.

### 2.1. CREATED NETWORK

In real-world conditions, it is typically found that users of email clients are connected to its corresponding “outgoing” server, the one responsible for their domain, which in turn is connected to all other existing servers and domains. Users will hand in their messages to its server, and then the latter will try to connect to the server in charge of the recipient's domain. From there, the messages will be delivered to the machines of their recipients.

In contrast, clients sending unsolicited bulk email, or “spam”, typically act behind their own dedicated server, and try to reach as many other mail systems as possible.

In order to recreate the aforementioned conditions, the script will automatically connect to the available machines specified in an input list, and install software on them until the following layout has been completed:

- A configurable number of “regular” email domains, each one featuring its own email server and antispam filter. This part of the deployment will use one or two machines depending on the filter configuration, as described in 2.2.
- A configurable but constant number of “regular” simulated email users, attached to each regular email domain. One machine per user will be used.
- A configurable number of spamming domains, each of them featuring a spamming server and a spamming user. Two machines per domain will be used.

Hereafter, this created network layout will simply be called “the testing system” or simply “the system”.

### 2.2. SUPPORTED SERVER-FILTER ARCHITECTURES

The way email servers hand in their messages to their associated filters can be configured into two different architectures: “piped” and SMTP.

The main difference is that a piped filter lacks the ability to communicate using the SMTP protocol, rendering it unable to receive remote emails, and thus it must be installed within the same machine as the email server - no independent filter machine is needed. Nevertheless, these filters allow for simpler email server configurations, as their

execution is handled by separate “local-delivery” software.

On the other hand, SMTP filters, while being able to operate from a machine which is not the same one that hosts the server, often need more complex configurations. They must receive the messages directly from the server, and then give back the result, using the SMTP protocol. This direct interaction demands dedicated server activity, hence the increased configuration complexity, but allows for more flexible network architectures and filter designs. Note that when SMTP operation has been configured, it is possible to install the filter in a different machine than the one hosting the mail server.

### 2.2.1. Piped filters

To use this kind of filters, the parameter *filtertype* from *setupvalues.cfg* must be set to *PIPE*. This mode of operation is preferred for simplicity, and is the default configuration for AntispamLab.

In our system, when an email server receives mail, it checks if the recipients are inside of its own domain. In such a case, the message is passed to “local delivery”, which in our tool is handled by the Procmail [4] software, in order to arrive to a user's mailbox. When PIPE functionality has been selected, Procmail is automatically configured to execute the filter and write the complete text of the email to the filter's standard input, and then listen from its standard output for the processed result. Finally, Procmail parses the result in search of the filter's verdict mark, and delivers it to a user mailbox accordingly. This full process takes place inside one machine, and only happens once per mail.

### 2.2.2. SMTP filters

To use this kind of filters, the parameter *filtertype* from *setupvalues.cfg* must be set to *SMTP*. In this case, each email server – Postfix [5] in our case - will have to handle two different SMTP mail sources: the filter machine and the rest of the network.

On the one hand, mails from the filter's machine will be received on a separate configurable port; these are the mails that have already been filtered. If their recipients are associated to Postfix's own domain, the messages will be handed in to Procmail, which will place them in their corresponding user's mailbox. Else, they will be passed to the destination's domain server.

On the other hand, mail from any other sources will be passed directly to the filter, using an SMTP connection to another configurable port.

As a result of the aforementioned procedure, all mail will be passed from the server to the filter, and after processing, back from the filter to the server, to finally be delivered to where its recipients are found. Note that there is no sensitivity regarding whether these recipients are inside or outside the server's domain; all mail will be filtered, whether it is outgoing or incoming. For this reason, if a mail is sent and the recipient's domain is different than the domain from which the mail was sent, two filters will process it: first when leaving the domain of origin, and again when arriving at the recipient's domain. While using AntispamLab, SMTP filter implementations must handle this particularity.

Filters using this architecture can be (re)started at the beginning of each testing run, but otherwise will not receive other commands. These filters are expected to remain active within their machines for the duration of the run.

## 2.3. STANDARDIZED FILTER INSTALLATION

AntispamLab automatically installs the filter to be tested in the corresponding machines. Since the installation procedure may greatly vary between filters, two standard files must be manually prepared first:

- A compressed gzip file, `filter.tar.gz`. It will be unpacked in the destination machine, and must contain the all the files that have to be used during the filter's installation. A common example would be a collection of `.rpm` files packed together.
- A shell script, `filter_install.sh`, that must contain all the commands needed to perform the installation and activation. The script will be executed with root privileges, and must run stand-alone - additional input or output data is not regarded. A common example would be a sequence of installation commands for the rpm package manager followed by command that executes the filter.

Additionally, some configuration parameters must correspond to the filter's actual characteristics:

- In `setupvalues.cfg`:
  - `filtertype` - Must be PIPE or SMTP, depending on the filter's architecture.
  - `filtercommand` - For PIPE filters only, specify the command that executes the filter.
  - `filterport` - For SMTP filters only, port from which the filter will receive all mail.
  - `filter_reset` - Command issued to the filter prior to each testing run. It allows resetting the filter, but can be set to NONE if no reset is needed.
- In `defaults.cfg`:
  - `DASKeyword` - Header (and value if needed) that the filter is going to use to mark spam. For example, SpamAssassin could use `X-Spam-Status: Yes` as the value for this parameter.

Filters that demand additional automatic configuration are not supported, and need to be manually installed and configured. Furthermore, filters are required to mark all found spam using a header, which must be also constant (although its value may change). Procmail is prepared to look for this header, and other methods of spam marking are not yet supported. Nevertheless, they would be still compatible if manual Procmail configuration is performed in the servers once the system has been deployed as normal.

## 2.4. SIMULATING CLIENTS AND USERS

For simplicity, both user activity and email client functionality are simulated by a single script, which creates and handles SMTP and IMAP activity. After launching a testing run, it is executed inside all "user" machines within each regular domain, and inside the "spammer" machine associated to each spamming domain. This said script will hereafter be called "user simulator" in this document.

### 2.4.1. User event schedule and time scaling

AntispamLab models email activity by the execution of two different events: "read" actions and "send" actions. While reading, users will check their mailboxes and inspect new mails. When sending, a user will pick up one random mail from its unique corpus division, and send it to another regular user in the system. Bear in mind that the same mail is never used more than once, so a big-enough mail corpus is a requisite in order to perform large tests, or otherwise a user may run out of unique messages. Nevertheless, this eventuality will be detected and reported as an "abort" message (see 2.6.1.).

In order to manage the simulation, the user simulator first creates an activity schedule, randomly filling it with events associated to the "nature" of the user. AntispamLab

currently features two different natures: regular users, which perform either read or send actions, and spamming users, which only perform send actions. This simple simulation can be easily improved by changing the source code, allowing for more realistic or varied activity patterns.

Event execution is programmed for specific times in the future, depending on two configurable parameters: First, the average distance between read or send events, which represents the amount of time that users take between performing actions through its email client. This average will be constant throughout the system, but can be set to different values for reading and sending. Finally, a time scale value, which will divide all other times in the system. Hence, a time scale of 100 would make the system's activity 100 times faster. This parameter allows time-effective testing of long-term features of the filter under test.

#### 2.4.2. Time overlaps

Nevertheless, bear in mind that PlanetLab's inherent resource scarcity will introduce delays in event execution that will not escalate with the time scale. Thus, time scale is limited by physical factors. This limit manifests through the hereafter called "overlaps", i.e. when a user-executed event takes so much time as to delay the execution of the next event in the user's schedule.

Although a few overlaps are common and not a major issue, if enough of them occur, statistics or repeatability of a test could be in jeopardy. Furthermore, when the mean execution delay is greater than the scaled average distance between events, the overlaps will appear subsequently and increase in severity. However, each user simulator will record all overlaps which exceed a configurable time threshold, and will prematurely halt execution if an also configurable number of them occur subsequently. In this case, the master machine will be informed as explained in 2.6.1, and using a lower time scale value is suggested.

#### 2.4.3. Collecting data

Whenever a user reads mail, i.e. executes a "read" event, it extracts the "Message-ID" headers (a unique mail identifier) from all new mail in its mailbox, and then checks them using a database of spam IDs. This database is generated automatically using all IDs from the mails in the provided spam corpus. Found ham is reported as a "true negative", whereas spam is copied to a "deleted as spam" mailbox and then reported as a "false negative".

Similarly, positives are reported after a user has executed its last scheduled event. Only then, it will also read their associated spam mailbox, which contains all mail that the filter had marked as spam. An ID check is performed, spam reported as a "true positive" and ham as a "false positive".

Once this last process is completed, all data is logged into a file and sent to the master machine.

### 2.5. DANGER SIGNALS

User feedback is a source of information that most modern spam filters exploit. Many commercial email clients feature some kind of "Delete as Spam" button, which lets an associated spam filter know that one of its judgments was erroneous. This feedback, bound to a specific user and a specific mail, will be called a "danger signal" hereafter.

Inside our system, the simulated mail client users will often use the IMAP protocol to read the messages in their inboxes. In the case of using an ideal spam filter, these would only contain ham (or legitimate mail), but what they will normally find is a mix between ham and spam, as certain amount of spam messages can pass undetected by non-ideal filters.

When users find a false negative, the corresponding message is spam and will be placed in a special mailbox. This operation is managed by the IMAP server, Dovecot [6], which has been patched to detect the event and perform the following actions:

- Extract the SMTP envelope of the mail
- Extract the recipient of the mail
- Execute an external script

The last step launches a separate process which is given the extracted data as execution arguments. The script provided to this purpose within AntispamLab will cleanup the data, and then send it to the filter's machine through a socket on a configurable port. Then, the filter can read from the socket and use the information to "train" itself. Additionally, when each testing run has finished, the same port will be contacted to request for a log by sending the string "SENDLOG". The filter can then either ignore this request or process it by answering with any amount of text, which will then be added to the global logs (to allow for any further external processing).

Please note that, due to time constraints, danger signal functionality has not been thoroughly tested and the default examples provided with the tool do not make use of it. In order to use it, a good understanding of the involved code is recommended. Nevertheless, customization has been expected, and all scripts concerned are independent. Changing *send\_danger\_signal.py* will suffice in most cases, but modifications in the Dovecot's patch or source code could be eventually necessary.

## 2.6. TESTING PROCESS

After a successful deployment, the created system is ready to perform tests. The tests are conceived as a number of "runs", or configurable-length periods of simulated email activity, that are launched with the execution of a script. Each run receives a different seed to feed the random generators in the system, which is in turn generated from a "master" seed given to the test.

To test a filter, the scripts in charge of simulating users will exchange mail among themselves. All messages are filtered upon arriving to their server of destination, and put into either the inbox or a special spam mailbox of the recipient. In addition, regular users will periodically read their inboxes and perform "delete as spam" operations if needed. Spammers will only repeatedly send different spam emails, each one addressed to a randomly selected regular user.

This activity takes place during a configurable amount of time, a run, after which the filter's decisions are evaluated. To reduce the impact of randomness in the results, each test involves a number of runs, which are performed using varying random seeds. Any parameters affecting a test, such as the time scale or duration, can be changed by editing the master machine's configuration files. When the full test is complete, "true positive" and "false positive" ratios are calculated, and 95% confidence intervals shown.

In addition, preceding each run, the run-launching script sees to resetting the status of the system. It will proceed to clear any remnants of previous runs that could still exist

within the network, in order to prevent them from affecting future results. The process includes emptying all mail queues in the servers, deleting the contents of the users' mailboxes, and issuing a configurable reset command to the filters.

When the aforementioned procedure has taken place, user simulators are contacted in parallel, their configuration files updated, and prepared for execution. Once all of them are ready, they are simultaneously instructed to execute, and email activity commences. A thread will listen in the background, receiving and handling any reports from remote machines. The run finishes after all users have reported.

### 2.6.1. User reports

Simulated users will report when one of these events occurs:

- All scheduled actions were performed, and a final mailbox check was successful
- An abnormal circumstance forced an execution halt

The first case represents the desired behavior, and will result in sending a text file with data on all the actions performed by the simulator. The main script will add this information to a global log file, which will be used to compute the results of the test. On the contrary, the second case only occurs when the script simulating a user has "aborted" due to detecting a critical error. Most commonly, "abort" reports originate from errors such as subsequent event overlaps or mail corpus depletion, but might be also caused by file errors and other events alike.

Once all reports have been received, the run is considered complete. Alternatively, if a certain amount of time is exceeded, a timeout will occur and the whole run will be discarded.

Finally, all available information is joined together inside a global log file. When this file could not be correctly created for any reason, an alternative file containing the word "error" will be generated instead, and will mean that all information from the last run is lost.

### 2.6.2. Filter reports

After collecting all logs from the simulated users, each filter, or any program listening on their behalf in a specific configurable port, will be issued a log request by receiving the string "SENDLOG". This way, if a response method is implemented, any information that the filter logs during runtime can be also collected. Nevertheless, this feature is never a requirement; if the response is not given, the corresponding filter request will be skipped and the test will continue normally.

Filter logs received this way will be merely appended to the global log file from the last run, but otherwise this data will be completely disregarded by AntispamLab. Any further processing desired must be implemented externally.

### 3. EXECUTION TIMELINE

Section 3 contains a high-level explanation of the tool's functionality timeline. The details can be found in the appendixes (7.1).

The tool's functionality sequentially develops in three main phases: "availability checking", "deployment" and "testing".

#### 3.1. AVAILABILITY CHECKING

The availability checking process was implemented in order to deal with the unpredictability of the status of the PlanetLab nodes specified in the input list of available machines. It basically identifies and marks unavailable machines from the list, in hopes that the ones remaining are enough to deploy a system.

This first phase takes place shortly after executing the main script. It begins with pinging all hostnames specified in *machines.txt*, and then saving their IP address, in the same order, in a new list, *ips.txt*. Whenever a host could not be reached, the word "unknown" is saved instead.

Once the pinging is completed, a small script is uploaded to all machines in *ips.txt*. The script then makes sure that the correct permissions are available in that particular host, and tries to contact back to the master machine. In turn, the master machine listens for these replies up to a set amount of time. When this time runs out, machines that have not reported are considered unavailable and discarded. Note that discarded machines are not directly removed from the input list file; they will only be internally marked and skipped by the tool.

#### 3.2. DEPLOYMENT

This is the second phase. It takes place after the availability checks and must be successfully completed in order to be able to perform any tests, since it creates the layout of servers and users by remotely uploading and executing all required software. The execution of *build.py*, which is the first step in the tool's usage, will first install all required python libraries, and then proceed to call *setsystem.py*. This latter script is the one in charge of the actual deployment process. It will perform any needed uploads and installations, and will finally create a file in the master machine, *results.txt*, that describes the network layout created. The file lists the action taken on each machine (server, client, spamserver, spamclient or error/removed) along with its hostname and IP address.

In addition, a feedback mechanism controls the aforementioned procedure. When a server installation script successfully completes, it reports back to the master machine, which in turn will be listening for 80 seconds. If the reply is not received within that time, a configurable amount of retries will take place, which will mean that the master script will re-upload all needed files and execute the server installation again. When the allowed number of retries have failed, the machine is skipped, and the installation takes places in the next available node. This way, saturated machines, which will not complete the installation in the given amount of time, are prevented from becoming servers in the tool's network.

Whenever a deployment fails due to lack of machines or other critical errors, it is possible to restart the deployment process in hopes of a better result. Nevertheless, the sever installation script cannot be run twice within the same machine, as part of its activity will overwrite or duplicate the existing settings. Hence, in the need of a re-deployment,

additional processing is required. This processing is also automatized in another script, *reuse.py*, which will be in charge of removing the previously-installed servers from the input list of available machines, so that a re-deployment does not use them.

The re-deployment procedure can also be used if an already-deployed, working system becomes undesirable, for example when needing to change layout parameters such as the number of spammers in the system, or after identifying slow nodes that are reducing the overall performance. The procedure is the same as after a failed deployment: making sure enough available machines are left in the input list, and running *reuse.py* and *setsystem.py*.

### 3.3. TESTING

Once the deployment phase has successfully completed, manual execution of *manyruns.py* is required. The initial seed and the number of runs to perform need to be specified as execution arguments. The main activity of this script is executing a loop which sequentially begins testing runs by invoking *runtest.py*. When the desired number of runs have finished, *manyruns.py* parses all logs received, calculates the filter's ratios via *extract\_results.py* and prints them to the screen.

## 4. DEVELOPMENT CHALLENGES

Even though I find that Python is notably easy to learn and use, the development of the tool's code was not exempt of difficulties. When it was finally decided to use PlanetLab instead of a virtualized LAN, many challenges arose; foremost among them was the realization of how far PlanetLab was from being an ideal network. Nodes went up and down or rebooted without notice, their resources eventually saturated, their configuration or software varied... Problems like these became evident in the first weeks of development.

As a result, one of my first conclusions was to assume that every host in the input list of machines had a high probability of being unreachable, and that reachable hosts had still a relatively low probability of being unusable. I handled both issues separately, first rejecting the nodes which could not be contacted and respond back, and secondly detecting and skipping slow machines during system deployment. These measures meant that bad nodes were automatically eliminated, and thus the probability of successfully running the tool increased with the length of the input list of nodes. This way, unsuccessful runs could always be solved by adding more machines to the list and re-running the tool.

Nevertheless, additional issues began to manifest while trying to configure a first version of a test server. Even though software installation was trivial, managing a node presented some challenges due to the restrictions of PlanetLab's virtual machines. Firstly, some standard commands, such as the one to add a user, are disabled. This forced me to devote some time to learning how the operating system worked, in order to directly edit the configuration files which are normally modified by the disabled commands. For example, in order to add the users to the server machines, instead of using *adduser*, the corresponding lines had to be added to the files */etc/passwd* and */etc/group*, using a specific format.

Secondly, most software assumes that its users have full control in the machine they are installed, which was not the case. For example, Postfix's error-logging mechanism relies in being able to access the operating system's logger process, not available in PlanetLab. Without the aid of any error messages, all problems in the email server installation had to be detected and identified using their consequences as the only hints. Needless to say, the associated debugging required substantial time. Fortunately, Postfix is a widespread solution and much documentation can be freely accessed through Internet. Thus, my error investigations were always accompanied of Google searches, which accelerated the process most of the times. Ultimately, my understanding of the software involved became strong enough to work-around all issues that were undocumented.

Additionally, name resolution proved to be a problem. The reason was that resolving the name of some PlanetLab nodes pointed to the wrong IP addresses. Apparently, these nodes had an external mail server who received the mail for all the LAN in which the node was physically situated. Hence, when Postfix tried to send mail to the node, it was automatically directed to the IP address of the external server, which was not a part of the tool's system. Needless to say, the intended recipients could not be reached this way. However, the correct IP addresses did not have to be resolved during run time, as they are constant and known after the system's deployment. Thus, I implemented an automatic mechanism that created a list of the correct addresses for each domain, and configured Postfix to skip name resolution by looking up at the file instead.

But PlanetLab is not the only one to blame for difficulties. For example, the base code that I received was not easy to re-use. Apart from user simulators, the rest of it was

simply too specific to the virtualized LAN architecture, which was the original environment for our tool, and proved of no use for an adaptation to PlanetLab. Nevertheless, it did give ideas and hints for many parts related to SMTP implementation and Python programming. Unfortunately, some reusable parts needed major revision and also were not completely bug-free. In some extreme cases, such as trying to reuse the filter simulator that was being used to check the testing process, re-use of existing code was finally discarded because of needing too much time to analyze and debug.

Furthermore, there were many other unforeseen problems that needed to be solved or worked-around in record times, especially with the time constraints imposed by related conference papers submissions. As an example, when accessing remote hosts via secure connections, the Python libraries that managed authentication keys proved to be ill-prepared for dealing with multiple parallel connections. That is, when many threads tried to save their new host keys to the master's machine file, they overwrote one another. To solve this issue, I had no other option than to access the related library, i.e. Paramiko (the one that provided methods to manage SSH connections), and modify its source code to make the key-saving method more robust.

Moreover, PlanetLab updated its nodes' operating systems while the tool was still under development, suddenly changing the installed software's behavior and requisites. The worst consequence was that Postfix's mailboxes stopped working, since the new operating system did not support the way they were configured at that moment (mbox style). That meant that all the mail for one user could no longer be stored in a single file, and therefore the mailbox style had to be changed to a directory-based model (maildir style). This change affected Postfix, Dovecot and Procmail, whose configuration needed to be reviewed and re-tested.

And lastly, I would like to mention that patching Dovecot was a task that required totally separated work. It had to be written in C, instead of Python, and Dovecot's source code has so many different files and method inter-dependencies that looked completely obscure to me. Even though some filter developers had already created similar patches, they were only suitable for a particular filter, and were not standardized in any way. In addition, their source code was not any clearer than the original. In the end, using some hints from all that I found in Internet, I was able to obtain a basic patch, functional enough to fulfill its task in a minimal way, but nevertheless lacking some of the originally intended features. Dovecot's patching proved too time-consuming as to continue its development, so I considered what I had as a work-around that could be improved in the future, and my focus moved to other endeavors.

## 5. CONCLUSIONS

After spending approximately half a year in developing AntispamLab, it satisfies me that the result is a completely functional tool. All the initially intended main features have been implemented, and although there is much room to further development, no loose ends have been left - the tool is completely independent and self-sufficient. In addition, new features were conceived, discussed and developed, including a standardized way of adaptation to the most usual filter software architectures. Moreover, whereas speed or efficiency have never been primary objectives, the main script is able to successfully run even in the often-hostile conditions of a PlanetLab network, starting parallel activities for speed and processing feedback for robustness.

Despite its complete functionality, this project should be considered as a settlement of the basis for a larger research. It is principally useful to begin investigating the differences, effects and characteristics of testing antispam filters within a totally realistic environment, as opposed to serially feeding mail to a filter. Additionally, many parts of AntispamLab's source code can be modified in order to increase the complexity of the testing system's behavior, which also represents new opportunities for experimentation. This further research, though, is beyond the scope of my thesis, and most of the tests that I performed were only oriented to checking the tool's functionality.

Nevertheless, the development of AntispamLab has been a highly enriching experience by itself. When I first heard about the project, I was just merely familiar with Linux and the way email/SMTP works, had absolutely no experience with SSH, Python, IMAP, or PlanetLab, and my programming skills were rusty. My work consisted in creating an upgraded version of a previous tool [1], but able to automatically spread and run through PlanetLab instead of using a single physical machine. Mainly, I revised the existing user/client simulators, created my PlanetLab-oriented system deployment process, and merged both things together. In addition, I implemented a number of new features in order to improve the testing process and automatize multiple testing runs. Finally, I developed standard methods to allow the general public to use AntispamLab without the need of specific knowledge, allowing the tool to adapt to the most common needs of filter testers by following very simple instructions.

Needless to say, my knowledge in the areas involved in the project has now greatly increased, as well as my programming ability in aspects such as multi-threading, system calls, sockets or even external code analysis. Additionally, I could also put under test my background as a system administrator, since many software installations and administrative tasks in Linux have been an important part of the project.

But these software and hardware skills are not the only benefit that I received from this project. Most importantly, developing AntispamLab involved a great amount of scientific methodology and critical thinking. Deciding between different courses of action, selecting the best solutions to a problem, pondering the feasibility of new features to implement, or setting development priorities have been crucial steps in the development, and I can safely claim that my capability in this regard has been proven, and that I feel comfortable to engage any similar future projects.

However, having to depend on many external factors, such as adapting to existing code or to PlanetLab's requirements, meant that some problems could not be solved directly. Hence, there are some parts of the code that are more a workaround than an accurate solution (e.g. circumventing a limitation of PlanetLab machines), and would make little sense if the tool were to be run outside PlanetLab.

In addition, I would like to mention that a variety of tools that facilitate interaction with PlanetLab are rapidly appearing nowadays. For example, some of them are devoted at monitoring node status, or at distributed software deployment and control. Being open-source, these third-party tools could prove suitable to merge with or to be adapted to AntispamLab, adding much more complexity to the features that are not directly related to testing a spam filter (such as the deployment phase).

Finally, I am proud to have chosen this project and brought it to fruition, since it looks like a complete work which can serve as the basis to interesting research. I am confident that the tool will be useful to the spam-fighting community, and hope it will be developed further in the future.

## 6. REFERENCES

- [1] – “System for Testing Antispam Solutions”, Raphael Naefen, EPFL. MICS summer-internship project report, October 2006.
- [2] – Planet Lab, official site: <https://www.planet-lab.org/>
- [3] – AntispamLab - A Tool for Realistic Evaluation of Email Spam Filters. Slavisa Sarafijanovic, Luis Hernández, Raphael Naefen and Jean-Yves Le Boudec. Accepted for CEAS 2007, Mountain View, 2-3 August 2007.
- [4] – Procmail, mail delivery agent originally designed and developed by Stephen R. van den Berg. Official site: <http://www.procmail.org/>
- [5] – Postfix, mail server by Wietse Venema. Official site: <http://www.postfix.org/>
- [6] – Dovecot, IMAP server by Timo Siranien. Official site: <http://www.dovecot.org/>

## 7. APPENDIXES

### 7.1. THE SCRIPTS

AntispamLab consists of a collection of separate scripts that run with a varying degree of independence. Only two of them must be manually executed, whereas the rest will be run automatically whenever they are needed. Each of these "master" scripts will control a main phase in the tool's operation; one will be in charge of deployment, which takes place first, and the second one will manage the testing process.

#### 7.1.1. Starting the tool

In order to start using the tool, a number of required files must exist within the master machine, all in the same directory:

- All files contained in the tool's package, *antispamlab.tar.gz*.
- A text file, *machines.txt*, containing the newline-separated hostnames of the rest of the nodes in the network (excluding the master machine) available to be used as part of the system.
- The RSA public and private keys, *id\_rsa* and *id\_rsa.pub*, that allow SSH access to the nodes specified in *machines.txt*, which have to be unique, i. e. they do not change between nodes.

Once these requisites have been met, desired configuration values must be set by editing *setupvalues.cfg* and *defaults.cfg*, using any text editor such as *vim*. Be sure to replace *filter.tar.gz* and *filter\_install.sh* with the ones corresponding to the filter to be tested.

The tool can be started by running *build.py* using the following command: *python build.py*. It will thereafter progressively output its status into the standard output, by default the screen, in form of regular text lines.

The script will, sequentially:

- Install python libraries in the master machine, associated with handling secure connections to remote machines.
- Cleanup *machines.txt* from whitespaces.
- Launch *setsystem.py -y*, which will start the deployment phase

The *-y* modifier will automatically assume an affirmative answer to any prompts, which is normally the desired operation. This way, the subsequent deployment phase will fully develop without user interaction.

#### 7.1.2. Deployment

In order to ensure the availability of the hosts listed in *machines.txt*, a number of other scripts are started by *setsystem.py* prior to deploying the system.

##### 7.1.2.1. Availability checking

Firstly, *pinger.py* will browse through the list of machines and translate each hostname into an IP address. A file under the name *ips.txt* will be created, containing either the IP addresses returned while pinging each host or the word "unknown" whenever a particular host could not be reached.

Next, *checker.py* is executed. It will start a new thread for each IP address present in the previously formed list; these threads will be in charge of managing an SSH (secure shell connection) to the remote machines. This connection will be followed by the uploading of a small script, *check.py*, which will ensure that a few basic commands can run and will report back to the master machine. Whenever *check.py* report back from a remote machine, the machine is considered usable and is recorded for deployment. If no answer is received after approximately 80 seconds, the machine is discarded.

Additionally, all SSH host keys (codes that identify a host) are handled in parallel, but in such a way that allows saving them into the master machine without overwriting or duplicating. This process has been implemented to avoid the need from user input.

Once *checker.py* has finished its tasks, the list of usable machines is saved into *okhosts.txt*, whereas their IP addresses are stored in *okips.txt*. Then, the actual deployment procedure is started by the execution of *setsystem.py*.

#### 7.1.2.2. Distributing and installing software

In order to create the desired network layout, *setsystem.py* first checks what the filter settings are. The simplest case is the one of a PIPE filter, since they use a standard mail server configuration. This means the mail servers from both spammer and regular domains will work in the same way, so that a single installation procedure can be repeated upon every domain. Therefore, procedure involves a loop with the following steps:

- Create domain-dependent configuration files for the server: the IP address of the machine that hosts the filter and the list of IP addresses of the domain's associated email users
- Upload server files to a host in the list of usable machines
- Remotely execute *set.py*, in charge of installing the server and reporting back to the master when successfully finished (more details in the next section)

At this point, still inside the same iteration of the deployment loop, *setsystem.py* will wait for an answer from the remote machine for a certain amount of time. If a timeout occurs, the upload and execution steps are retried the number of times specified in *setupvalues.cfg* under the parameter *maxtries*. If no response is received once all retries have taken place, the remote machine is considered unusable and skipped. Otherwise, once an answer is received, server installation in the current domain is complete, and the master script proceeds to take further steps (still for the same domain):

- Create domain-dependent configuration files for the clients and filters, namely the IP address of their associated email server
- Upload filter files to the same machine on which the server was installed
- Remotely execute the filter installation script
- Create user-dependent configuration: seed, IMAP username, email address, and type (either spammer or user)
- Upload the user simulator files to the next machine in the list

This last step is repeated once for each user that needs to be created in the domain. Once this is completed, the deploying iteration finishes and the loop proceeds with the next domain. It then repeats the aforementioned steps until all of the domains have been deployed successfully.

Alternatively, when SMTP filters are selected, some changes apply to the

aforementioned deployment procedure. The difference lies in the fact that regular and spamming domains no longer share the same email server configuration. The reason is that regular domains need to interact with its filter using SMTP, which demands a more complex server installation. Hence, the script separates regular servers from spamming ones, and modifies the installation process accordingly by uploading specific pre-made configuration files.

Nevertheless, although the uploaded files change, the only high-level effect is that, in the case of selecting SMTP, the machine hosting the filter does not necessarily need to be the same one that hosts the email server. It is also possible, depending on the parameter *external\_filter* in *setupvalues.cfg*, to deploy it to the next IP address in the list of usable machines. Note that user simulators will always be uploaded to the IP addresses in the list that follow the one corresponding to the filter.

### 7.1.2.3. The server installation script

The script which creates a domain server is *set.py*. It is uploaded to the corresponding machines and then remotely executed, as mentioned in the previous section. For debugging purposes, its standard and error outputs are redirected to *log.txt* and *err.txt*, which can be inspected by logging manually to the machine.

When executed, its first concern is to make sure there is no other already-running instance of the script, i.e. it has not been executed more than once. This can eventually happen if the installation process takes too long, since the master machine can detect a timeout and start the script again. If any running python process is detected, the execution of *set.py* is aborted; in any other case, the installation procedure begins. Note that future modifications of AntispamLab must be aware of (or remove) this feature if other python processes must be present on the server at the time of installation.

If no other python threads are detected, */etc/services* is modified in order to change the default SMTP port to the one specified in the tool's configuration. Bear in mind that, within PlanetLab, ports below 1000 cannot be used, which means that the standard value of 25 is not allowed.

Next, the script proceeds to modify */etc/passwd* and */etc/group* in order to create the accounts associated with email users in the domain. Their home directories are prepared and all needed permissions are given. Once the accounts have been successfully created, all required software - Postfix, Dovecot and Procmail - is installed.

After completing the aforementioned preparations, the main configuration process takes place. It involves the following (listed in order of execution):

- Giving each user the same configurable password
- Configuring Postfix's domain-dependent parameters: setting the domain name as the machine's hostname, pointing to the transport table to use and pointing to the list of IP addresses of the allowed users in the domain
- Modifying Postfix's configuration files in order to adapt to the selected filter type
- Modifying Procmail's configuration files and replicating them to all the users, in order to specify the header to look for when an email has been marked as spam, the name of the users' mailboxes and the command to run the filter (if it is PIPed)
- Starting Postfix
- Preparing Dovecot's configuration files for operation through a configurable port
- Starting Dovecot

Finally, the script connects to the master machine in order to report the successful

installation. This report consists solely of the word "FINISHED", no further details are given.

#### 7.1.2.4. Post-deployment

In a deployed system, there are a number of configuration parameters which must refer to the final network layout, such as the list of mail addresses of all users or the IP addresses of the servers in their domains. Nevertheless, the complete layout cannot be foreseen until deployment has finished, since the IP addresses that will host the servers are unpredictable (due to installation failures forcing to skip machines). Thus, these parameters cannot be configured beforehand, while the deployment is still in progress.

For this reason, once the software for servers, users and filters has been installed in all their corresponding machines, *tables.py* is executed. This new script first reads the created network layout from *results.txt* and creates a list of the IP addresses associated to the server of each email domain. This list, called "transport table", will be used by each server in order to be able to reach all others. Once this is done, *tables.py* will upload to each user its specific configuration file, which will contain a list of the recipients that the user can select for its mail. The recipient list does not include the addresses of spamming users or each client's own address.

Finally, once the execution of *tables.py* finishes, *divide.py* is started by the master. This last script creates a database of the Message-ID headers of all mail in the spam corpus, which are saved into *spamids.txt*. Then, it handles the separation of the provided email corpus into smaller unique parts which will be uploaded to the users in the system. In order to create these parts, folders corresponding to all users are created, and then each provided mail is randomly placed within one of the folders. When the process is completed, each folder is compressed and uploaded to its corresponding machine along with *spamids.txt*. This procedure takes place twice, separately for spam and ham.

#### 7.1.2.5. Re-deploying

A new deployment over an already-deployed network cannot be performed by directly executing *setsystem.py*. This is due to the fact that the server-installation scripts cannot run in a machine that is already a server. Hence, removal of the previously-installed servers from the list of available machines is required. Nevertheless, this removal is also automatized, and can be performed by entering the command *python reuse.py -a* in the master machine's console. Alternatively, entering the same command without the *-a* argument will not only remove servers, but also all machines that were considered unusable during the previous deployment phase (due to having failed availability checks). This latter option allows skipping the availability checking phase when redeploying the system.

When *reuse.py* has finished, the redeployment can be started by the command *python setsystem.py -y*. This will restart the procedure explained in 3.1.2 and 3.1.3. The *-y* argument is actually used to automatically assume 'yes' as the answer to some user prompts implemented in *setsystem.py*. Although this is normally the preferred behavior, as it allows the script to fully execute without user interaction, the argument can be omitted. In this case, *setsystem.py* will ask the user whether he wants to perform the availability checks or skip them, prior to automatically deploy a system. Skipping the checks can save time when the machines in the input list are already known to be reliable (due to having been checked previously, for example).

Note that the previous system will not be completely cleaned when redeploying. For

example, the script that simulates the users will not be deleted, and will remain in some machines even though they might not be “user” machines in the new system, and therefore never execute the script. However, apart from the existence of these innocuous residual files from the previous system, a redeployed network is equal to a newly-deployed one, but just using different machines.

### 7.1.3. TESTING PROCESS

Once a complete system has been successfully deployed, filter testing can commence. A test is managed by *manyruns.py*, which is executed by entering the command *python manyruns.py* plus two more numerical arguments: the number of runs in the test and the initial seed with which to feed random generators. For example, *python manyruns.py 20 1* will start a test composed by twenty runs, using 1 as the initial seed. The initial seed will be used to generate a random list of sub-seeds which will be given to each run in the test.

After generating the sub-seeds, each run starts with the execution of *runtest.py*, passing a sub-seed as an execution argument.

#### 7.1.3.1. Network reset and run launch

The first concern of *runtest.py* is to reset the system's status, deleting all eventual remnants from previous runs. The script will read the network's layout from *results.txt* and perform the following actions:

- Connect to all servers and empty their mail queues, deleting any leftover messages
- Connect to the servers corresponding to regular domains and empty all their users' mailboxes, including spam mailboxes
- Connect to all users, both regular and spamming ones, and execute *updater.py*.
- *updater.py* remotely changes the configuration file of each user to match the one present in the master machine. This update will only affect the parameters *duration*, *timescale*, *time overlap threshold*, *sequential overlap threshold*, *logfile*, *seedvalue* and *read/send time averages*.
- Connect to all filters and execute the configurable reset command (see 2.3. for details)

Once these steps are successfully completed, the users are contacted again, this time in parallel, and executed simultaneously once all of them have been reached. The email activity will begin and thus the run is started. Then, *runtest.py* will proceed to listen to a configurable port in order to receive the log from each user simulator.

#### 7.1.3.2. Log collection and results

If the run progresses correctly, the users should finish their execution after a delay of *duration / timescale* seconds. A small amount of uncertainty still exists, since user simulators schedule their last action at a random distance from the maximum run duration, and this last action is in turn affected by execution delays and time overlaps. Nevertheless, this uncertainty pales in comparison to how machine status and resource availability can affect the process. Mainly, a user simulator might take too much time in sending its log when its machine's bandwidth is saturated, meaning that the log collection process can take twice the time it would when under ideal conditions.

Although even huge such delays could be tolerated, they can mean that the test will last much longer than intended, and a re-deployment of the system might be a good idea. For this reason, when 5 times the expected delay have passed and there are still users which have not reported, a timeout will occur and the whole run is discarded. The timeout value

is hard-coded and bears no other significance than being big enough to be almost certain that a major problem has occurred while trying to receive a log.

If the timeout is not exceeded, each received user log is appended to a global log file. The global log files are specific to a single testing run and are automatically saved and re-created by *manyruns.py*, so that a series of files like *globalLog1.xml*, *globalLog2.xml*... *globalLogN.xml* is created after each full test.

The data logged this way is in the form of xml entries that contain information on the activity of the user simulator that generated the log. It includes true/false positives/negatives, which are used to calculate the filter's ratios, and also additional data, such as the identities of the messages that the user sent or read. A *.xslt* style sheet is provided with the tool, and can be used to manually inspect a particular log using any xml parser, such as a simple web browser.

Next, the script contacts all filter machines and sends a "filter log" request as described in 2.6.2. Any response will be added to the global log, but this is never a requirement for the filter; hence, non-responding filters will be skipped without any further consequences. These filter logs can be used for any additional processing that the users of our tool could need.

Then, *runtest.py* finishes its execution and control returns to *manyruns.py*, which will rename the current global log to the form *globalLog#.xml* and finally proceed to start the next run. When all intended runs have been completed, i.e. *manyruns.py* exits its main loop, *extract\_results.py* is executed. It will parse all global logs, searching for all xml entries that contain TP or FP *type* fields (standing for true and false positives) from last test and show its associated ratios. The number of logs that could be successfully parsed is also shown, by the name of "statistically relevant".