Stamp
A Mailing List Manager for Squeak

Anselm Strauss
strauss@iam.unibe.ch

May 10, 2007

Software Composition Group
University of Bern
http://www.iam.unibe.ch/scg

Contents

Introduction 4

1. Architecture and Design 5
- 1.1. Overview . . . 5
- 1.2. Mail Handling . . . 6
- 1.3. User Interface . . . 8
- 1.4. User Management . . . 8
- 1.5. Software Dependencies . . . 8

2. Implementation 9
- 2.1. SMTP Service . . . 9
  - 2.1.1. The SMTP Protocol . . . 11
  - 2.1.2. SMTP Commands . . . 12
  - 2.1.3. Server States . . . 13
  - 2.1.4. Messages . . . 13
  - 2.1.5. Exception Handling . . . 13
  - 2.1.6. Limits and Checks . . . 14
- 2.2. Core Model . . . 14
  - 2.2.1. List Manager . . . 15
  - 2.2.2. Lists, Users, Subscriptions and Contacts . . . 15
- 2.3. Concurrency . . . 17
- 2.4. Configuration . . . 18
- 2.5. Web Interface . . . 18

3. Issues 22
- 3.1. Storage . . . 22
  - 3.1.1. MailDB . . . 23
  - 3.1.2. Magma . . . 23
- 3.2. Concurrency . . . 24
- 3.3. Dynamic Configuration . . . 24
- 3.4. Data Overflow . . . 25

4. Future Work and Extensions 27
- 4.1. Message Archive . . . 27
- 4.2. User Interface Enhancements . . . 27

4.3. User Roles 28
4.4. Bounce Processing 28

Conclusion **29**

Acknowledgments **30**

A. Software Dependencies **31**

B. Installing and Setting up Stamp **32**

Introduction

Stamp is a Mailing List Manager entirely written in Squeak[1]. Although it falls in the same category as other mailing list managers like Mailman[2] or Ezmlm[3] it should not be regarded as equivalent. Such applications have undergone a long development time that is beyond the scope of this project. And still, there are a lot of shortcomings reported by users: complicated installation, non-intuitive web interfaces, ugly code that is difficult to enhance and so on. Stamp tries to pick-up the whole thing from beginning and implements a clean base that can be extended.

This project's focus lies in the experience that can be gained during implementing a mailing list manager in Squeak. Especially the experience that can be made when applying object oriented design to real applications, and also the experience from applying Squeak or Smalltalk in general to application focused projects.

[1] http://www.squeak.org/
[2] http://www.gnu.org/software/mailman/
[3] http://www.ezmlm.org/

1. Architecture and Design

This chapter will give some information about essential characteristics of Stamp that are responsible for why Stamp is made the way it is. They determine Stamp's behavior and its capabilities. Some characteristics are forced by technical reasons or software requirements. Others are based on decisions, like design decisions or implementation choices.

1.1. Overview

Figure 1.1 presents different aspects of Stamp to show their main appearance in context of the whole system. For an overview Stamp can be split into two different parts. One covers all mail handling, which includes receiving, queuing and sending mail. The other part deals with the core model for users, their contacts and lists. Most of Stamp's functionality can be assigned to one of these parts.

Mail is received upon a new connection to the listener from outside. Each new mail spawns a new handler that is queued and processes the mail message. Incoming mail is called a *posting* to a list. Ideally the only thing that must be done with a posting is to look up the recipient list and forward the message to all subscribers of that list through the sender component. The connection to the outside is usually realized via a single SMTP server.

The most heavily used component is the manager, called list manager, although it not only manages lists but also users, contacts and more. It owns different collections with the model data and coordinates concurrent calls from all over the system. It also performs various checks for all calls to preserve consistency of the data.

Apart from the mail handler processes there are no other processes running in the background. All action is initiated through either incoming mails or by a user over the web. The web interface is implemented with Seaside[1].

[1] http://www.seaside.st/

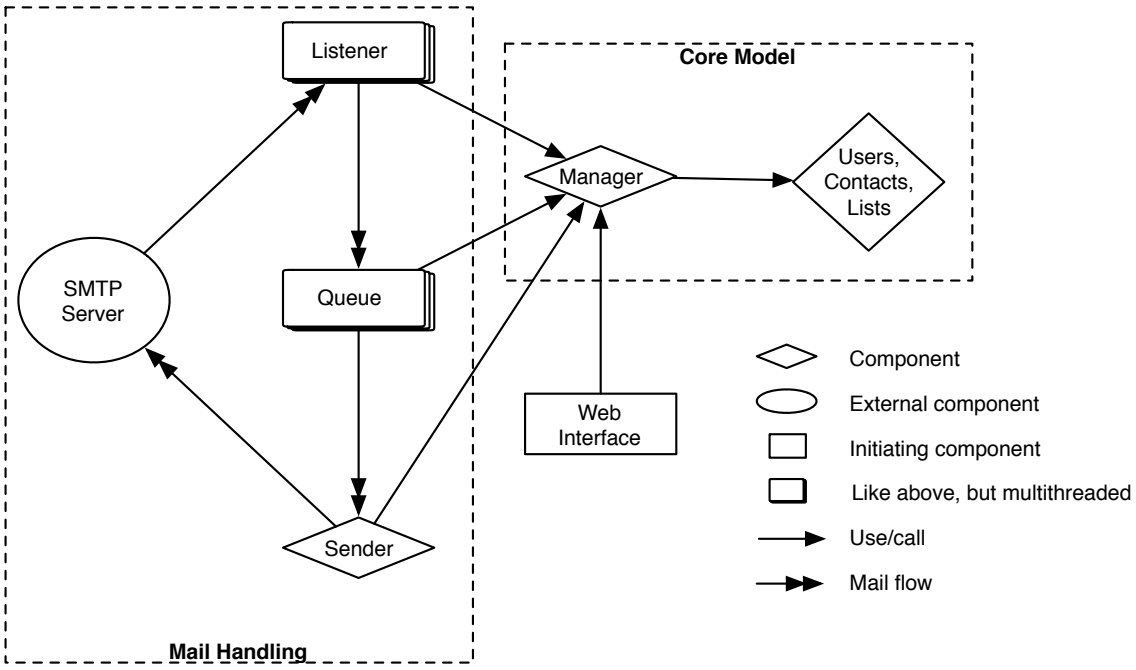

Figure 1.1.: Stamp overview

Since more than one list server per Squeak image is usually not required, and to avoid the difficulties of running multiple servers within an image concurrently, Stamp is implemented as singleton. `SPKernel>>instance` will return the only living instance.

1.2. Mail Handling

An important part of a mailing list manager is mail handling. One question that appeared in the very beginning of the project was how to get the mail into the system. Some list managers use hacks in the mail server code or in the configuration to intercept the normal mail flow and instead pass the mail to their application. This has the drawback of adapting the software for each server software out there. It also relies on some sort of software compatibility between the two programs. It can be difficult to hack into other environments, like C binaries or a shell script, from within a Squeak image. Furthermore such a hack most probably requires both systems to reside on the same host. In some scenarios it might be vital for security and performance to be able to separate the systems.

A more expensive but also cleaner solution is to interface the mail with SMTP, as is done

for example between mail relays or spam mail hubs. This eliminates the drawbacks described above. It increases the independency, compatibility and also the mobility of the software. Stamp is in fact an SMTP server with minimal capabilities.

Figure 1.2 shows Stamp's usual network layout. Stamp is located inside a secure network. It uses a single SMTP server to sent mails out and receive new ones from it. This server has full SMTP functionality and is responsible for relaying the mail further to the internet and routing incoming mail for the mailing list manager to Stamp.

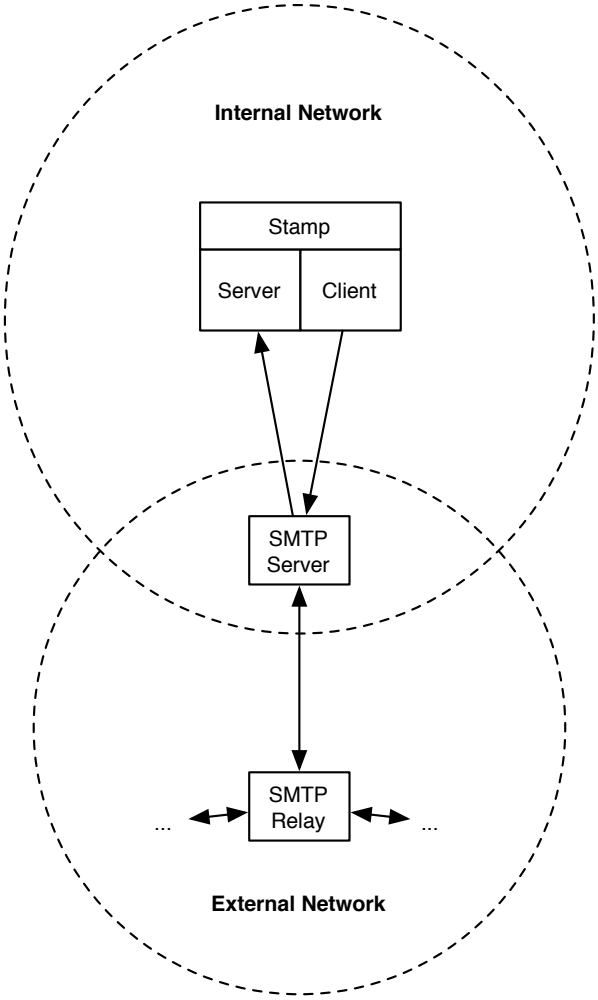

Figure 1.2.: Stamp SMTP overview

1.3. User Interface

Traditionally list managers can be controlled by a dialog-like mail conversation with a special command syntax. Modern list managers also support a web interface that can be much more comfortable. For Stamp the decision was made to use only a web interface. With a good web application framework like Seaside this is the easiest way to implement a user interface.

1.4. User Management

A mailing list manager does not necessarily need a user management. It is sufficient if it identifies mail addresses. However, this has the disadvantage of not properly reflecting how users really use the list manager. For example, one user might have multiple addresses. It is not very intuitive to have a separate account with a separate password for each of these addresses.

For Stamp the decision was made to use a proper user management with minimal functionality. Stamp's user profiles include a username with a password and any number of contacts. A contact is nothing more than a mail address that can be referenced by a subscription. Subscriptions connect contacts and lists.

1.5. Software Dependencies

Stamp has a number of software dependencies apart from the standard packages installed in the Squeak default image. Table A.1 in the Appendix gives some information on the requirements and the tested versions.

2. Implementation

This chapter will give some more technical insight into the most important components of Stamp and how they are implemented.

2.1. SMTP Service

`SPMailListener` is a wrapper for `SPMailService`, which is itself a subclass of `TcpService`. There is little functionality in these two classes. Each connection is handled by its own instance of `SPMailReceiver` where the real work is done. Figure 2.1 shows and UML representation with the most important relations, attributes and operations.

The listener manages a list of all connections to limit the number of concurrent clients if necessary. It can be started and stopped. The mail service inherits from `TCPService` and spawns a new process for each new client. `SPMailService>>serve:` is called with a new `Socket` and the control is entirely passed to a new instance of `SPMailReceiver`, allowing the mail service to return quickly and be able to respond to other clients[1].

The receiver makes a lot of use of its neighbor classes. It calls the listener to register and unregister the connection, creates SMTP commands that operate on itself and passes on received messages to the queue. As long as neither the server nor the client requests to quit the session and there are no connection interrupts, the receiver traverses an endless receiving loop. It can either be in command mode, which will be most of the time, or data mode. It implements the ordinary command pattern as described in [1].

[1] In early versions of the code the listener, service and receiver were split up into three separate classes since they all inherited from different superclasses. With the current code base at least two of them could be merged into one class, but the code was not yet adapted.

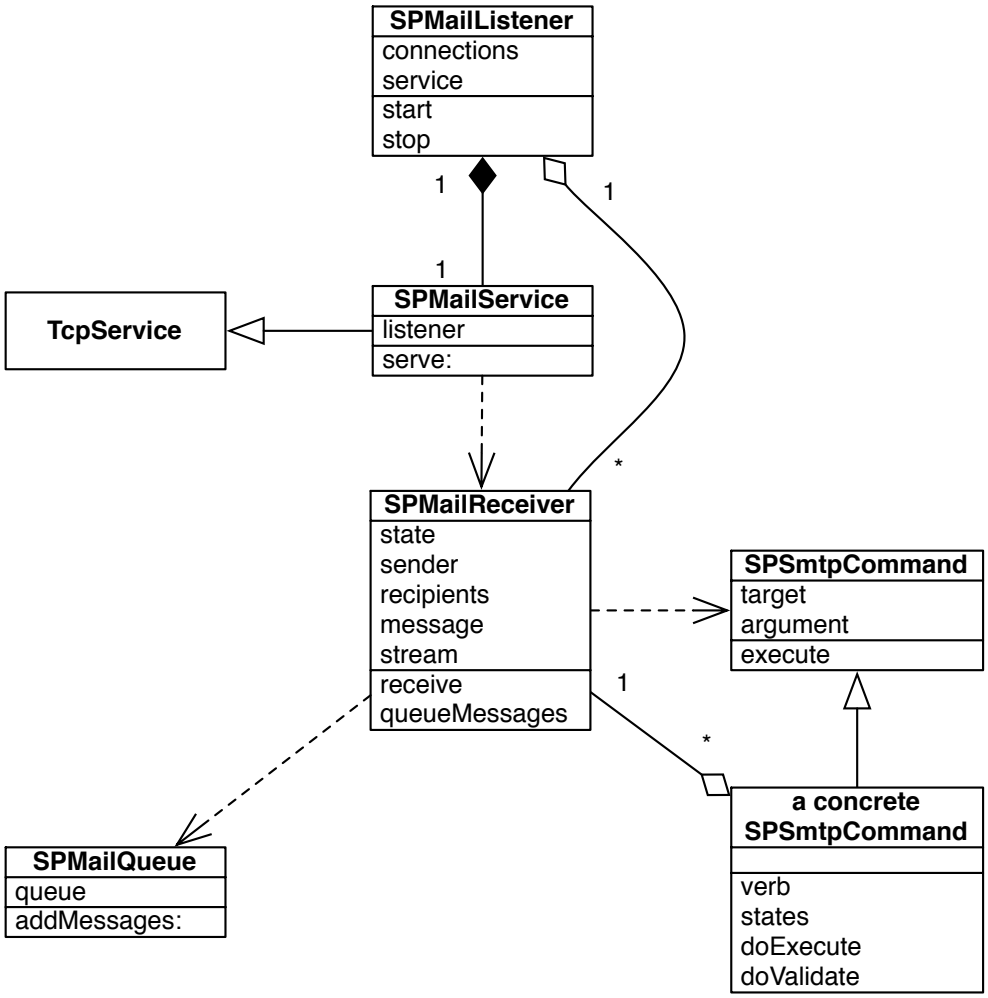

Figure 2.1.: UML diagram of the SMTP service

2.1.1. The SMTP Protocol

SMTP is basically a Telnet session but with limited syntax, so the conversation is rather simple. A complete documentation of the SMTP protocol can be found in [2] and [3]. The following shows a sample conversation between a server and a client. Lines marked with `C` are entries made by the client, `S` denotes the servers reply.

```
C user@client ~ $ telnet server 25
  Trying 1.2.3.4...
  Connected to server.
  Escape character is '^]'.
S 220 server ESMTP Postfix
C helo client
S 250 server
C mail from:<me@domain>
S 250 Ok
C rcpt to:<you@domain>
S 250 Ok
C data
S 354 End data with <CR><LF>.<CR><LF>
C Subject: Re: Thesis
C From: God
C To: Nietzsche
C
C Nietzsche is dead!
C - God
C .
S 250 Ok: queued as 778L112XZ73
C quit
S 221 Bye
  Connection closed by foreign host.
  user@client ~ $
```

Such a session can be done with a simple telnet client. The client usually sends the commands `helo`, `mail from:`, `rcpt to:`, `data` and `quit` in this sequence. The server replies to each one of these commands with a number and a message. The numbers have all a special meaning and make it easy for a program to evaluate the session. One interesting thing to notice is that a separate sender and one or more separate receivers are specified in the SMTP session, apart from the ones in the mail message. They are used by the servers when relaying mail and are usually not visible in the users mail client application. In the example above arbitrary names for the `From:` and `To:` fields in the message header were

used. Normally these are checked for valid addresses by the accepting server, but they are not responsible for how a message is relayed.

The receive methods in `SPMailReceiver` are responsible for handling all SMTP conversation with the client. They are a bit special, since they try to protect against data overflow when for example a client is trying to spam the server with garbage. This issue is discussed in chapter 3.4.

2.1.2. SMTP Commands

The receiver maintains a state, to verify the right order of client commands, and some buffers for sender and recipient addresses, and also for the actual message. In command mode it tries to extract a command from each received line with `SPSmtpCommand class>>commandFor:target:`. This searches the subclasses to find a command whose verb matches the beginning of the line. An optional argument is given the located command upon initialization. The command gets executed on its target (a receiver), which involves validating the state, validating command specific things and finally running the code in `a concrete SPSmtpCommand>>doExecute`. Each command returns a collection of states, so it can be verified if it is permitted to execute in the current state of the receiver. Each command should also set a reply and a new state, if it does not do so default values are set. After successful execution of the command, the receiver sends the reply to the client, enters the new state and is ready for the next command. The implementation of a concrete command is very simple and does not require a lot of code, as the example of the complete code of `SPResetSmtpCommand` shows:

```
SPResetSmtpCommand>>doValidate
    "nothing to do"

SPResetSmtpCommand>>doExecute
    target reset.
    self state: SPResetSmtpState.

SPResetSmtpCommand class>>states
    ^ Set new
    add: SPResetSmtpState;
    add: SPMailSmtpState;
    add: SPRecipientSmtpState;
    yourself.
```

```
SPResetSmtpCommand class>>verb
    ^ 'rset'.
```

Everything else is implemented in `SPSmtpCommand`, the common superclass of all SMTP commands.

2.1.3. Server States

The states of the receiver can easily be implemented as symbols. But in Stamp they are implemented as classes. These are actually empty, but have some advantages compared to plain symbols. First, they can have a class comment that is in the right place to document the state. And second, it's easy to search for references to it by using the built-in *"class refs (N)"*-function of the context menu in the system browser.

Each command returns a set of states to describe in which receiver states it can be executed. Like this the receiver verifies the right order of commands. Upon successful execution each command returns a new state the receiver transitions to.

2.1.4. Messages

A message to more than one list is split up into multiple messages, one for each list. Messages should be treated independently for each list, so errors for one list don't affect other ones. Success is only reported when all messages have successfully been added to the queue. The queue spawns its own processes for each new message, since it can take multiple attempts until the message is forwarded to all subscribers and the client doesn't need to wait for this.

2.1.5. Exception Handling

The receiver and all commands can throw exceptions at any time, as long as they preserve consistency within themselves. Exceptions are caught on top of the calling stack in the receiving loop. A receiver exception inherits from `SPReceiverException` or `SPSmtpException`, where the latter inherits also from the former. They are initialized with the appropriate receiver and as default action just send a reply to the client operating directly on the

receiver. Any unknown exception, e.g. an exception not of kind `SPReceiverException`, causes the session to be aborted and displays as *"System error"* or *"Unknown error"* to the client. This allows the server to quietly go over unexpected errors and to continue normal operation, but demands special intervention when debugging.

2.1.6. Limits and Checks

In data mode input from the client is not interpreted but just buffered up to a special escape sequence which denotes the end of the data transmission. The received data is interpreted as ASCII text and converted to Squeak text. This involves converting carriage returns, line feeds and special SMTP escape sequences.

There are a number of limits and checks that are performed on the client. Even when they make the server more stable and secure it is not recommended to directly expose the server to the internet. It should rather be run in a private network or even on the loopback. A real SMTP server can easily be configured to forward mails for a specific sub-domain to the Stamp SMTP service and to accept sends from it. It can either be run on the same host as Stamp or on a separate one. The performed checks and configured limits include:

- Maximum number of concurrent clients
- Maximum number of recipients
- Valid addresses for sender and recipients
- Recipient is an existing list address
- Command length and mail data limits
- Queue full

2.2. Core Model

The core model of Stamp is designed for minimal but complete functionality. In short: there are lists, users, contacts and subscriptions. A new user registers with a unique name and a password. Everyone is allowed to register in the web interface. He adds some contacts, verifies them and subscribes to one or more lists. At any time he can activate or deactivate his contacts or subscriptions, so he can pause the reception of postings. He can

remove individual subscriptions and contacts, or unregister which removes all his contacts and subscriptions automatically. There is only one administrator, a superuser account, usually named *root*, that can manage pending subscriptions, add or remove lists and do more. Lists can have a separate administrator address, where notifications concerning this list are sent to. Otherwise all notifications are sent to a globally defined address.

2.2.1. List Manager

As figure 2.2 shows, the core model is heavily focused on `SPListManager`. The reason for this is that most operations need to be synchronized, for example Squeak collections are not thread-safe. There are also a number of Stamp specific things whose consistency must be preserved. Additionally most operations on one component of the model make use of different other components. Rather than create high coupling between classes it was chosen to centralize most operations into one class, even when this one gets a bit big.

The list manager owns some sets of type `Set` and `IdentitySet` depending on if the contained objects need to be distinguished by equality or identity. There are two special things to notice in the core model: subscriptions and contacts.

2.2.2. Lists, Users, Subscriptions and Contacts

As mentioned before, Stamp keeps apart different users and not just addresses. A user can have any number of contacts, but contacts can't be used right away for subscriptions after adding them. First they must be verified. On initialization of a new contact a random verification code is generated which is sent to the address of the contact. If the user really registered a contact that he is able to receive mail for, he will be able to read the code and verify the contact through the web interface. If he registers any random address he can't verify it which prevents misuse of addresses.

Lists can either be locked or not. A unlocked list allows anyone to post to it and also anyone can subscribe to it. In contrast a locked list allows only postings of subscribers and a requested subscription must be accepted by the administrator.

In the case of subscriptions, which are basically only a relation between contacts and lists, pretty much the same way as the relations between users and contacts, the relation is implemented as separate class not just as a reference. This is justified by the fact that

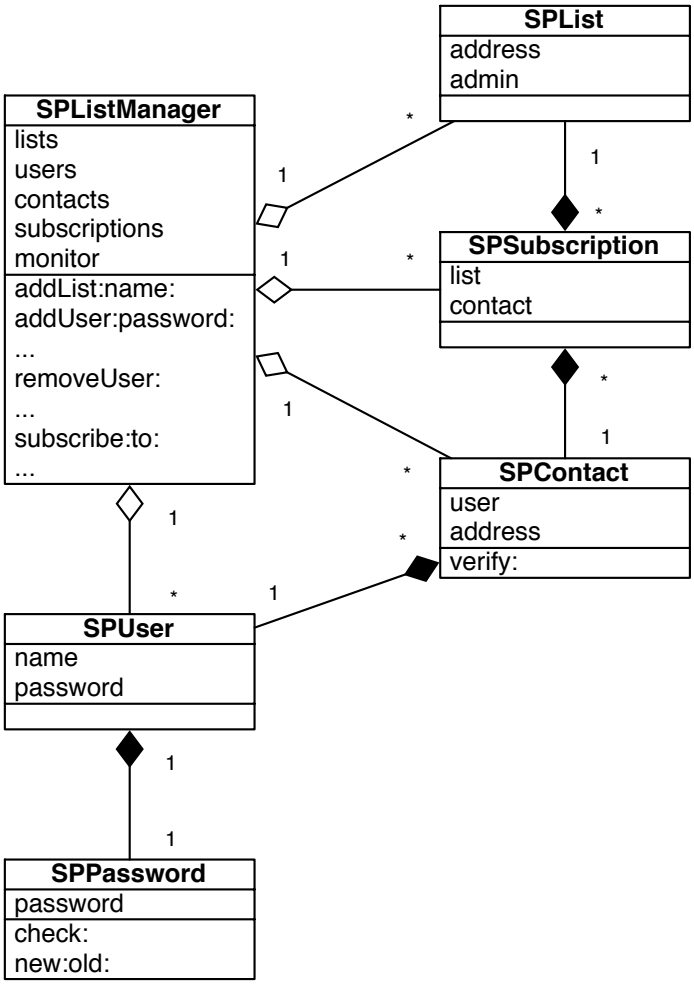

Figure 2.2.: UML diagram of the core model

each subscription has an additional attribute to state if it is active or not.

Subscriptions, contacts and user passwords fully depend on their containing class. The list manager automatically removes them if the container is removed.

2.3. Concurrency

There are different parts of Stamp that must be prepared for concurrency. All are protected through monitors of type `Monitor`. Squeak monitors are very useful, actually the only needed method is `Monitor>>critical:`. Very simple usage of monitors is made in the listener and message queue code. More advanced concurrency considerations are needed in the core model. There are three main issues:

1. Squeak collections are not thread-safe. All operations for manipulating and also for reading them must be protected with a monitor. Other operations, e.g. the process of checking the availability of a user name and registering the new user, are not atomic and must be done inside the monitor as well.
2. One can request the list manager to return a whole set of objects. Even when it is guaranteed that the set and its objects are not modified outside the manager, it is not a wise idea to just return a reference to the set that is internally used for all operations. If the set is used while the manger manipulates it, this can lead to inconsistencies for the reader. Not only write operations must be mutually exclusive but also read and write operations. Therefore the manager returns a copy of the set. Of course the process of copying must also be protected by the monitor.
3. When throwing exceptions within a monitor it must be ensured that the monitor is left as soon as possible to not block other operations. This can happen when not resolving the exception by either aborting or resuming it. One example is the standard debugger that appears by default for most exceptions. Another problem is Seaside with its continuation driven design. When going back in browser history when a debug page appears, the exception seems to stay unsolved and the monitor is locked until the debugger is opened and closed again. *In the case of the list manager this can block a lot of other operations. The manager uses only one monitor for all synchronized operations.* This might not be best for performance but avoids deadlocks that can easily arise with multiple monitors.

Beside the list manager no other class of the core model uses monitors or any other sort of locking. They are kept simple, so that no operations need to be synchronized. For example

assignments are atomic.

2.4. Configuration

Several preferences of Stamp need to be adjusted at runtime. Stamp uses a simple configuration system. When initializing the Stamp kernel a default configuration is created. It keeps a dictionary of name-value pairs. The name is a symbol and the value is a `SPConfigurationAttribute`. Configuration attributes do nothing more than wrapping around a specific class and perform special checking when setting new values. All attributes check for the right type when setting a new value and are able to take the value from a string. `SPIntegerConfigurationAttribute` for example is an integer that can be initialized with lower and upper bounds. It is used by `#listenerPort` which is a TCP port and must be in the range from 1 to 65536.

The configuration is actually not accessed directly but through an instance of `SPConfigurator` that allows concurrent access. Configuration is central and accessed by many components, either by a given reference on initialization, or statically by using `SPKernel instance configurator`.

2.5. Web Interface

The implementation of the web interface with Seaside is very straightforward. The following figures will give some ideas what can be done over the web interface.

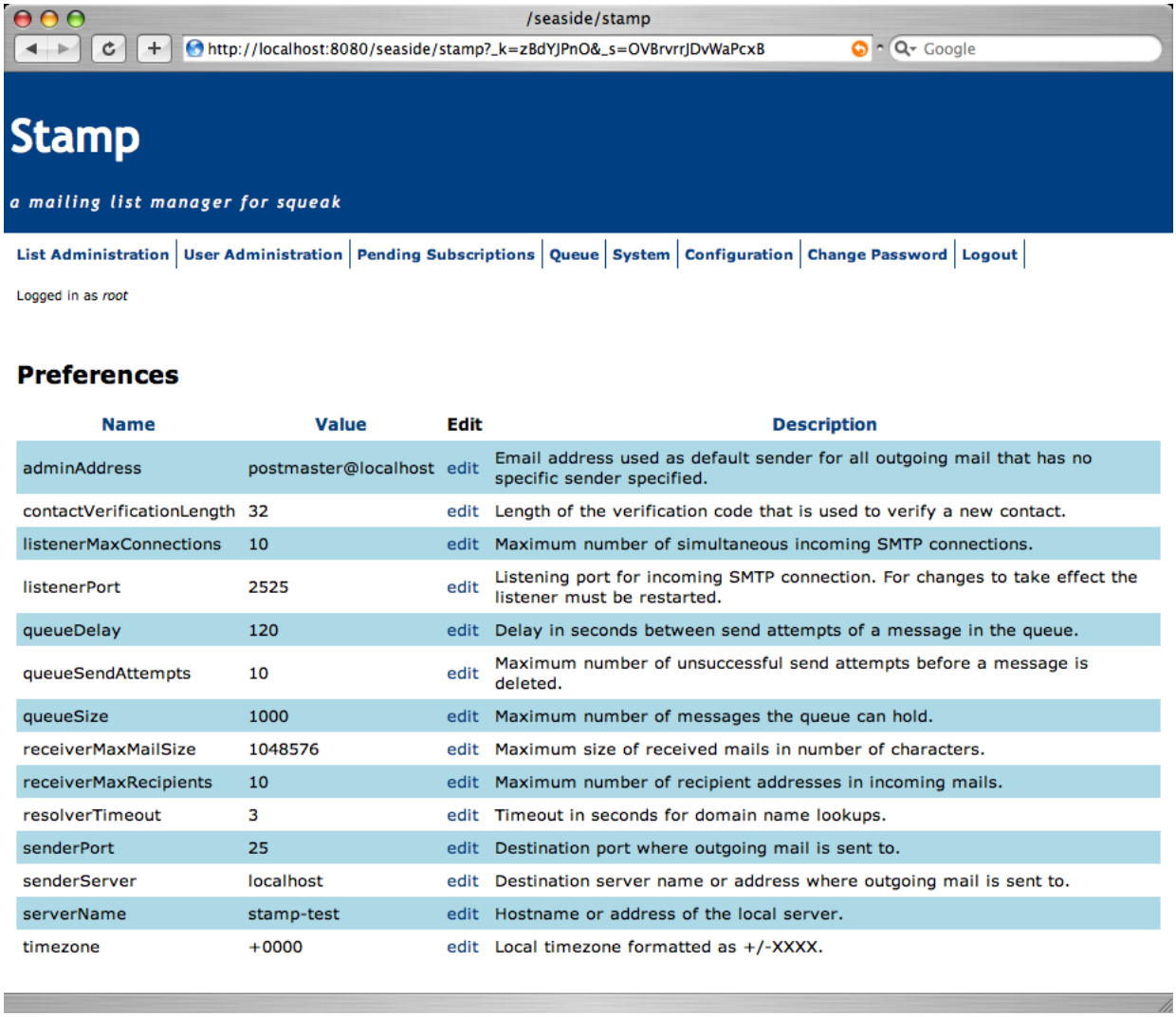

Figure 2.3.: Configuring Stamp through the web interface

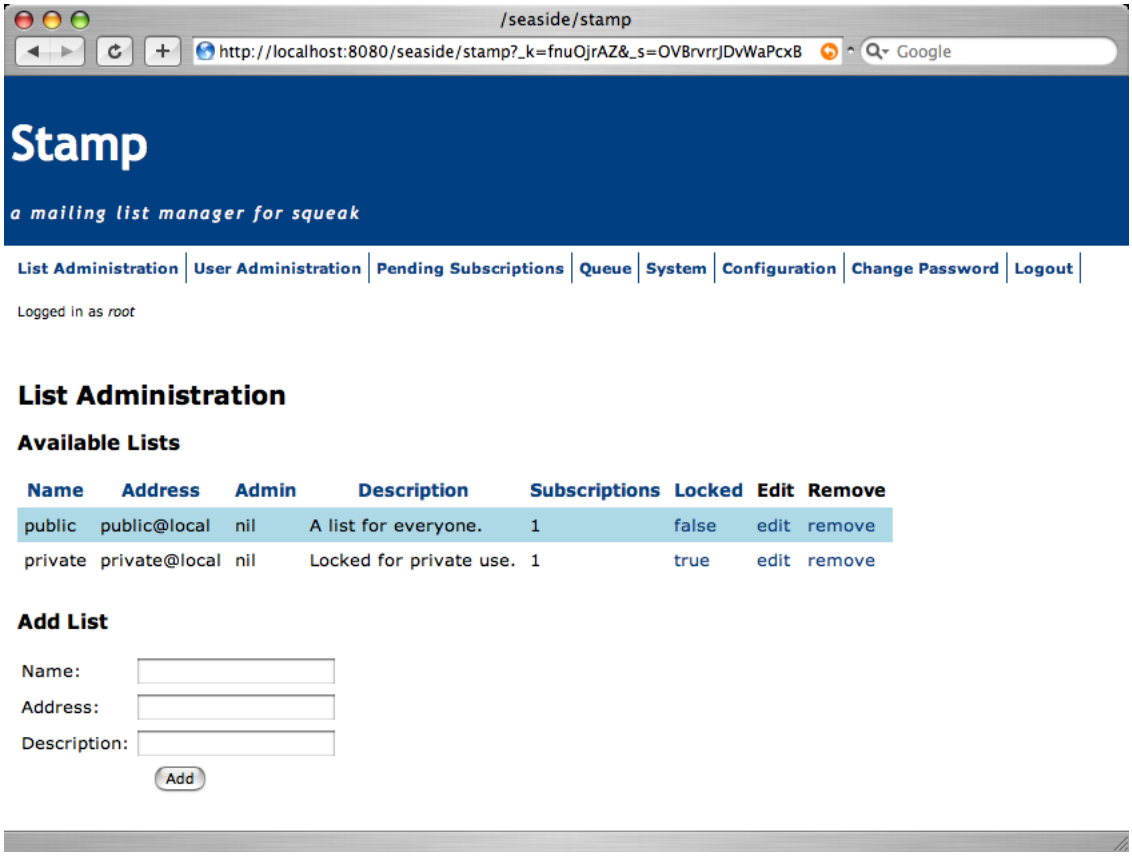

Figure 2.4.: Managing lists

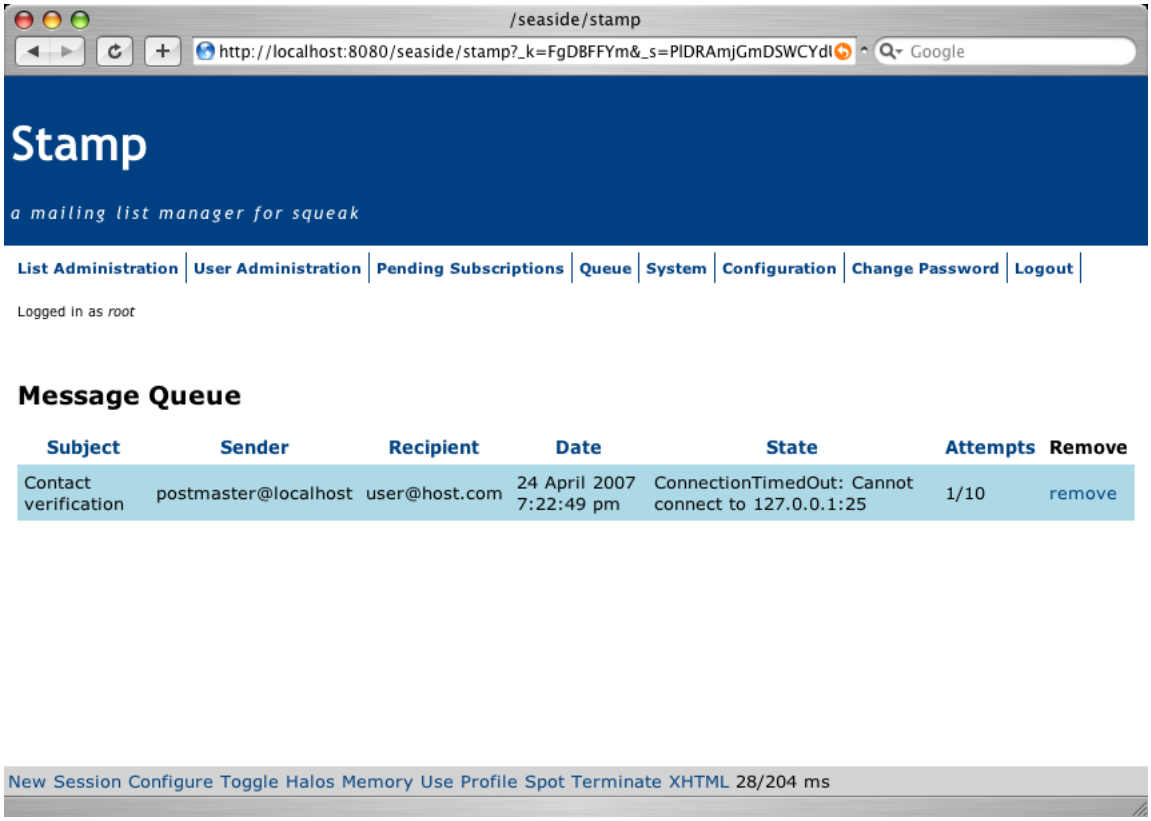

Figure 2.5.: Displaying Stamp's message queue

3. Issues

This chapter will focus on some problems that emerged during the project. Whether they were solved or not the issue stays and is worth mentioning.

3.1. Storage

There are two sets of data that Stamp produces: model data like users, lists, etc. and archived messages. Stamp stores the model data directly in the Squeak image for now. A message archive is not yet implemented, also because of various issues involved with storage.

Squeak actually does a good job by storing everything as object data in the image. Nevertheless there are a few concerns with this. First, this storage format is not well, or even at all, supported by other software. What to do if the image crashes and is unrecoverable or one wants to port data to a new image? There is surely not a lot of software that can access a Squeak image directly and retrieve or export the needed data. Second, a message archive can grow a lot, quickly exceeding a computer's main memory. Squeak would need to have an intelligent caching algorithm to operate on an image that does not fully fit into the memory. And besides, the image seems to be the wrong place for data that is not live, but rather seldom accessed.

There are two candidates to solve the storage problem. *MailDB* and *Magma* were tested during the project. But to mention it right away, neither of them was really satisfactory. The reasons are given below.

3.1.1. MailDB

MailDB is part of the *Celeste* mail client for Squeak. It offers an easy way to store instances of `MailMessage`, even when the handling of relative filenames is buggy[1]. This solution seems easy and cheap, but is insufficient. It only solves the part of mail message storage but must be adapted for everything besides `MailMessage` and also for all modifications of it. Furthermore it doesn't really solve the problem of caching, since each message that is accessed is just loaded into memory not caring about free space.

3.1.2. Magma

A far more complete solution is Magma[2], an object database for Squeak. It's a sophisticated project that definitely covers the essential features needed by Stamp, including:

- Transparent access and easy integration
- Consistency by transactions and fault tolerance
- Uses intelligent caching and querying
- Concurrent multi-user access

Magma works generically on objects thus allowing to store theoretically any sort of data. There is no need of converting objects, they are stored as they are, as objects. Instead of retrieving and committing with a database client all the time, objects are once registered and modified inside a commit block. Operations on objects stay the same, changes are done automatically on the database. When using big sets of data, like message archives, intelligent querying allows to just load the objects really needed. Caching enhances performance and allows to limit the amount of memory used, whatever size the database really has.

Although Magma seems to be a perfect candidate for Stamp, there are (yet) two major issues with Magma:

1. Opening a new session is expensive and usually takes a few seconds to initialize. Each new web or SMTP client needs a new session which would soon become a performance

[1] It is not possible to use relative filenames with multiple directories in `MailDBFile>>reopen` and `MailDBFile>>save`

[2] http://wiki.squeak.org/squeak/2665

problem. As workaround sessions can be opened in advanced to reduce the waiting time for new clients but not really enhancing performance.

2. When adding new objects to the database, they never get cleared from the cache until the session is closed and a new one is opened. There must be some logic that decides when to close and open a new session, so that neither the cache is overfilled nor too many sessions are opened which is, as said before, very expensive.

However, this issues seem to be solvable and known to the Magma community. The chances are good that a future version will fit all of Stamp's needs. But for now the problems stay unsolved and Stamp does not have a real storage system.

3.2. Concurrency

The only part of Stamp that is a real concern for concurrency is the core model. As shown in chapter 2.2 the model is relatively simple and so are the design decisions described in chapter 2.3 to preserve different concurrency aspects. What showed up to be difficult is to not break various concurrency guarantees like safety or introducing deadlocks when changing the code. Beyond that, it is even more difficult to design code so that breaking its concurrency guarantees during code development is less likely or at least doing so is noticed clearly and early. A lot of the code does just work in the special way it is used in Stamp, but would break if used in a more generic way.

As in general with concurrency it is hard to test it. Most problems appear at runtime and often first when the system is used in a production environment.

3.3. Dynamic Configuration

Stamp's configuration can be changed at any time during runtime and the changes are propagated automatically, even though not instantly but over time. This is not self-evident. In the case of Stamp this is achieved by simply requesting preference values each time they are used. For example, the queue is not initialized with a specific size, but its size check queries the configuration each time the preference is used. Of course this lowers performance a bit but saves code for observing configuration changes. Another drawback is that the configuration is heavily referenced and used through all the system.

Some code must be explicitly written with the fact in mind that some values obtained from the configuration can change between any two calls. An example is again the message queue. For each newly added message it is first checked if there is enough space. The check method is:

```
SPMailQueue>>checkFreeQueueSize: aNumber

    (queue size + aNumber) > self queueSize ifTrue: [
        SPQueueFullException signal.
    ].
```

Of course this works also if the actual size has already exceeded the queue size. In early versions of the code only one message was added at at time, and the check was `size = queueSize ifTrue: [ "throw exception" ]`. This is okay if you know that the queue size never changes. But when suddenly the queue size is set below the actual size the test returns false even though the queue is actually overfilled.

3.4. Data Overflow

It lies in the nature of Squeak to give the developer a lot of freedom and keep things as easy as possible. This is what makes it a well suited language for research. But as shown in the following example this can be a problem when Squeak is exposed to an untrusted environment.

The SMTP service of Stamp operates on `SocketStream`. This one has methods like `upToAll:`, `nextLine` and others that receive and return data up to a specific delimiter but return only when the delimiter is reached. In combination with Squeak's implementation of strings that can grow indefinitely, this presents in the context of untrusted clients a security flaw. A client that sends an arbitrary string which does not contain any of the defined delimiters will spam the server and exhaust its memory. It's the same problem as the Comanche web server has, which is often used in combination with Seaside. A simple command line on a UNIX shell that uses netcat, like `yes | nc <server> <port>` is sufficient to exhaust the server's memory and produce an out of memory warning. By the way of comparison the Apache web server closes the connection as soon as it recognizes that the client request can't form anymore a valid request.

That's why Stamp uses its own receiving methods. It receives characters one by one from the stream with `SocketStream>>next`. As soon as the received data exceeds the limits the

connection is not closed but the buffer doesn't grow anymore. This is achieved by checking the received amount of data all the time. The following code shows the core receiving method:

```
SPMailReceiver>>receiveTo: aString maxSize: aNumber

    "Receives all data up to and including aString.
     After receiving if the data exceedes aNumber
     an exception is thrown, otherwise the data is returned.
     Receives but avoids storing data if aNumber
     is exceeded to prevent overflow."

    | data buffer overflow |
    data _ buffer _ ''.
    overflow _ false.
    [ data endsWith: aString. ] whileFalse: [
        buffer _ self receiveNext.
        overflow ifFalse: [
            data _ data, buffer asString.
            overflow _ data size >= aNumber.
        ] ifTrue: [
            "add buffer to data but cut at start to prevent overflow"
            data _
                (data copyFrom: buffer asString size + 1 to: data size),
                buffer asString
        ].
    ].
    overflow
        ifTrue: [ SPTooMuchDataException signal. ]
        ifFalse: [ ^ data. ].
```

This solves the problem of exhausting memory on the server. In combination with the ability in Stamp to limit the number of concurrently connected clients this gives a good control of limiting clients. Even if Squeak is normally secured by some proxy servers one cannot eliminate the problems like that. It doesn't need to be a client's bad intent, it's enough if some system is behaving incorrectly.

4. Future Work and Extensions

This presents a list of possible extensions to Stamp and is by far not complete. The following are just suggestions.

4.1. Message Archive

One common feature of mailing list managers that Stamp is missing is a message archive. Due to the storage problems and lack of time this was not implemented. It can be kind of simulated by gathering and filtering all messages in a mail client. But usually it's important for new subscribers to browse old messages, search them and even retrieve old ones. This functionality should also be integrated into the web interface.

4.2. User Interface Enhancements

There are certainly a lot of things that could make the user interface, meaning the web interface, more comfortable, speedier and also more powerful. Some examples are:

- Searching for lists, users, ...
- Updating views on each page load and reload[1]
- Full control with super user, e.g. modifying individual users
- Connecting different views, e.g. clicking on an address in the subscription list should load the contacts page
- Operations on multiple items, like mass subscription

[1] Currently table data is only updated when the page is opened again, because sorting information of the table is lost when updating it. This is a bit uncommon since browser reloads do not update data. This is a problem in `WATableReport`.

One good example for a really sophisticated user interface is *Dabble DB*, but unfortunately the software is not free. A demo movie and application can be found on http://dabbledb.com.

4.3. User Roles

Stamp users don't have different roles. There is one special user, the super user, identified by its name which is normally *root*. He is allowed to do everything, whereas normal users can just operate on data they own. An administrator of a list is just a mail address that is used as sender, so that bounces and error reports can be split up per list. But administrative operations still need to be done by the super user.

At least it should be distinguished between the super user, administrators or managers for individual lists and normal users that are just subscribers. More flexibility could be reached with a true permission management like ACLs. There are a lot of imaginable access rights, like subscribing users, changing system configuration, editing list parameters, receiving notifications, etc.

4.4. Bounce Processing

Bounce processing presents a challenge because the list manager needs very good capabilities in filtering and assigning incoming bounces to the appropriate problem. There is no standard format of bounce messages which makes it difficult to identify the problem of an incoming bounce. One mechanism to solve some of these problems is called *VERP* (Variable Envelope Return Pathes) and described on http://cr.yp.to/proto/verp.txt. Each sender address is added to the original recipient, so when a bounce comes in the original recipient can automatically be determined.

Such things require the list manager to keep a history about all messages sent out. It must have a good notification system to inform users about failures. It must also be protected against misuse of bounces. And at least it seems almost impossible to automatically handle all bounces, so bounces need to be passed up to be handled by users themselves.

Conclusion

It was clear from the very beginning of the project that a lot of things would not be implemented because of lack of time. There are actually many things that need to be looked at closely if you want to make such a software really reliable. They may not look very complicated in nature, but bringing them into good object-oriented, self-explaining, short, maintainable, tested and clean code is still a difficult task.

Squeak has proven to be good at quickly reaching some minimal working code. It helps and even forces the developer to focus on the actual architecture and design of a software. It keeps away implementation details and avoids bothering with language specific problems. One can concentrate on the core and the real problems. On the other side there is sometimes too much freedom and with it too much clutter in the system. It's difficult to establish your own rules if you know that everything is dynamic and can be bypassed in some way. A lot of things are not explicitly documented. Of course you can always read the source code and write tests, but this won't give you the same certainty. Squeak clearly seems to be made more for testing environments than for production environments.

Acknowledgments

I want to thank Marcus Denker[2], who supervised me through the project, for his help and patience, when things were not moving forward so fast as they maybe should have.

[2] http://www.iam.unibe.ch/%7Edenker

A. Software Dependencies

These are the required packages and the versions of them that have been successfully tested with Stamp. Newer and older versions may or may not work. The packages should be installed in the order they are listed in the following table.

At the time of writing this document the packages were not yet listed as being compatible with Squeak 3.9-7067 or have not yet been published for Squeak 3.9-7067, but they did work stable enough anyway.

Package	**Tested version**	**Reason**
Squeak	3.9-7067	-
Dynamic Bindings	1.21	Web server dependency
KomServices	1.1.2	Web server dependency
KomHttpServer	7.0.4	Seaside dependency
Seaside	2.6a	Web interface is implemented in Seaside
Cryptography	0.3	Encrypting user passwords
Vassiliś Regex	1.1	Syntax checking of mail addresses, IPs, etc.

Table A.1.: Stamp software dependencies

B. Installing and Setting up Stamp

The following assumes some basic knowledge of Squeak and SMTP. This documentation is also available in the `SPDocumentation` class of Stamp's source code.

1. First, you need a Squeak installation for your platform and a clean Squeak default image to start with. Squeak is available on http://www.squeak.org/.

2. Install the software dependencies described in chapter A. All of them can be installed through the SqueakMap package loader as shown in B.1.

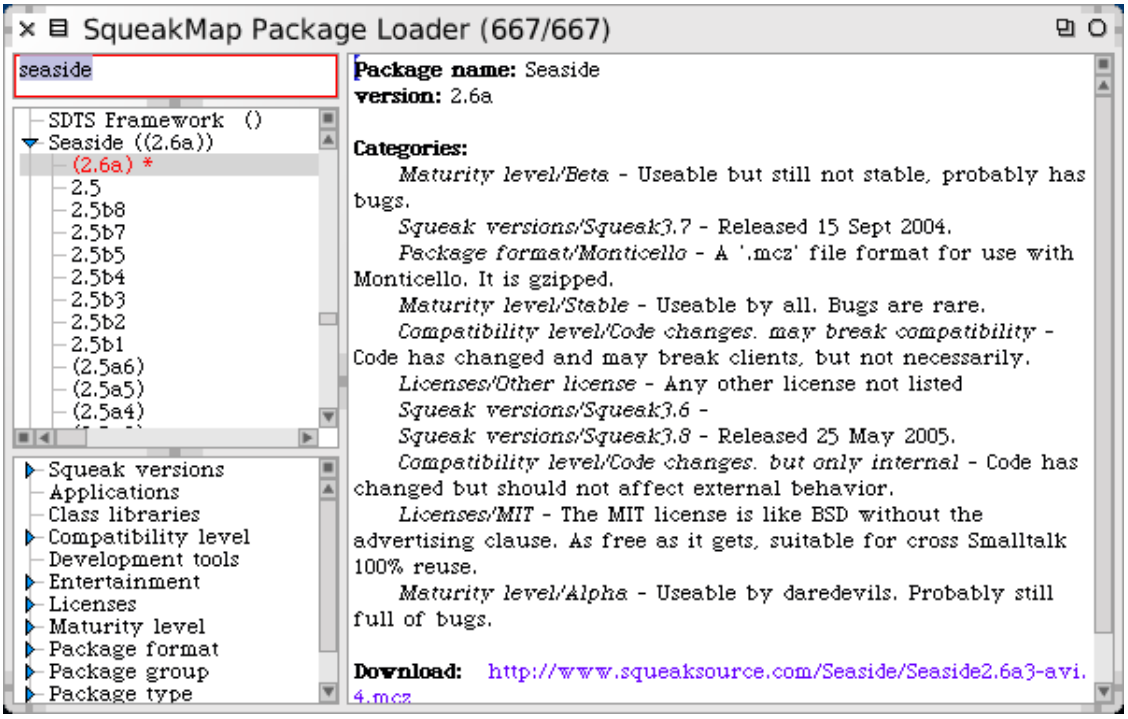

Figure B.1.: SqueakMap Package Loader

3. Stamp itself is available on http://www.squeaksource.com/. The code can directly be loaded into a Squeak image with Monticello.

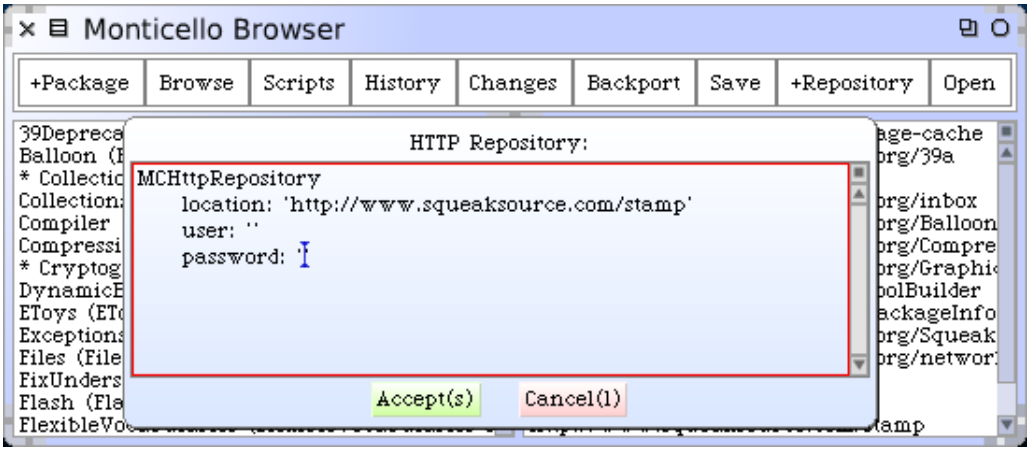

Figure B.2.: Monticello Browser

4. Start Seasides web server with something like `WAKom startOn: 8080` and register Stamp's web application with `SPWeb initialize`.

5. The Stamp web interface should now be accessible on `http://localhost:8080/seaside/stamp` as figure B.3 shows it. You can login with the super user account. The login name and the initial password for the super user are defined in the class `SPListManager` in the methods `superUserName` and `superUserInitialPassword`. Normally the super user name is *root* and the password is *changeme*.

6. After you have logged in you should *change the super user password*, adapt Stamp's configuration to your needs via the configuration tab and restart the listener via the system tab.

7. At last, configure your SMTP server to forward all mail to Stamp that is destined for list addresses you defined in Stamp. Usually you will configure the server to route all mail for a specific subdomain like `lists.mydomain.com` to the Stamp server. Also make sure your SMTP server accepts incoming mail from Stamp and relays it properly. The configuration for this depends on the software that runs on your mail servers. You may need to ask you mail administrator to make the appropriate changes. For example in *Postfix* this is documented in the *"Postfix Basic Configuration"* README file in the sections *"What clients to relay mail from"* and *"What delivery method: direct or indirect"*. An online version is available on http://www.postfix.org/BASIC_CONFIGURATION_README.html.

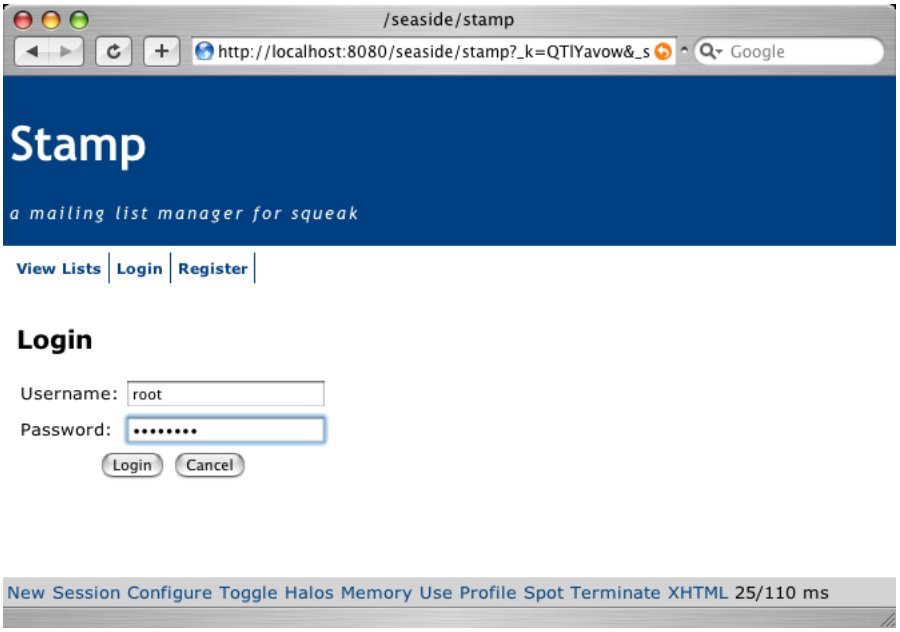

Figure B.3.: Stamp Web Interface

Bibliography

[1] Sherman R. Alpert, Kyle Brown and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley Longman, 1998.

[2] J. Klensin. *Simple Mail Transfer Protocol*, RFC 2821. The Internet Society, 2001. http://www.rfc.net/rfc2821.html.

[3] P. Resnick. *Internet Message Format*, RFC 2822. The Internet Society, 2001. http://www.rfc.net/rfc2822.html.