PostOffice Specification v0.1

Dillon Dixon

August 30, 2013

Contents

Chapter 1

Introduction

PostOffice is a library designed for interprocess communication via sockets to a *Post Office* server daemon. Clients register a password protected *mailbox* with the server daemon, which can be used to send and receive mail to and from other client mailboxes.

There are many existing message passing solutions, such as *DBus*[1] and *CORBA*[2], but most are highly generalized and intended for large scale deployment, often resulting in hefty documentation and buggy language bindings.

PostOffice was created in response to those developers who want a simple, platform independent and intuitive way for passing messages between processes. The whole system uses the common terminology of the actual physical mail system, making it immediately familiar.

This document functions as a description of the underlying protocol for those interested in implementing a *Client* binding for a different programming language.

[1] http://dbus.freedesktop.org
[2] http://www.corba.org/

Chapter 2

Architecture

2.1 Overview

PostOffice functions, as defined by its name, like a post office. An abstract description of the functionality can be found in figure 2.1 below.

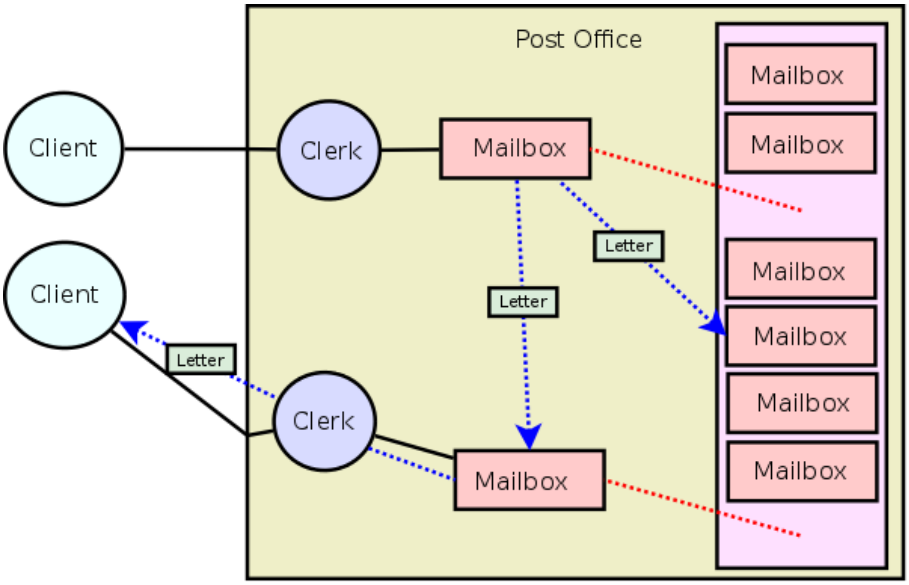

Figure 2.1: Abstract Diagram of Post Office

Clients connect to a *Clerk* (a thread on the server to handle the particular client connection). Clerks perform commands server side on behalf of the client, which involve sending and receiving mail, among other operations. When a client connects to a clerk, they must bind to a particular mailbox. If one does not exist, the clerk can create one for them. Clerks can also destroy them when they are no longer needed. Clients can create and destroy as many mailboxes as they like.

Mailboxes are the identifiers for clients in the post office. All mailboxes have unique identifiers. Like a physical mailbox, a mailbox can receive and store messages for a particular client while they are disconnected. A mailbox may be checked out by only a single clerk at a time.

Once a mailbox is checked out, a client can send and receive messages via the clerk. Any outgoing mail will be recorded as having originated at that particular mailbox. Mailboxes are password protected to allow only their creators to use them. The protocol by which actions are performed through the client is documented in later sections.

2.2 Control Flags

The following control flag enumerations are used for communication between the client connector and the post office server. Each code is represented by a single byte when sent over the socket.

The following list of flags in table 2.1 are commands that the client can make to the server to request actions.

Table 2.1: Client requests to the server

Name	Code	Description
REQBOX	0	*Request box* – Checkout an existing mailbox.
RETBOX	1	*Return box* – Return an existing mailbox.
CREATEBOX	2	Create a new mailbox.
REMOVEBOX	3	Delete/destroy an existing mailbox.
SENDLETTER	4	Send a letter to a mailbox.
GETMAIL	5	Retrieve mail from the connected mailbox.
NEXTLETTER*	6	Request the next letter in the connected mailbox.
SATIATED*	7	Request for no further message transfers from the connected mailbox.
EMPTYBOX	8	Empty the connected mailbox.
DISCONNECT	9	Indicate a disconnect from the post office server (impeding socket closure).

Request flags marked by an asterisk (*) are *non-initiating* flags, meaning they cannot be used as immediate requests to a server, and only as part of another request.

The following list of flags in table 2.2 are responses the server can make to client requests.

Table 2.2: Client requests to the server

Name	Code	Description
Name	Code	Description
REQGRANTED	0	*Request granted* – Indicates the successful completion of a request.
REQDATA	1	*Request data* – Indicates the client should now send some data.
NOAUTH	2	Made a request which the client is not authorized to perform.
BADCOMMAND	3	Received an unknown command code from the client (possible sync issues).
ALREADYCONN	4	The client is already connected to a mailbox.
MAILTIMEOUT	5	The waiting period for new mail has expired.
BOXEXISTS	6	A box was requested to be created, but it already exists.
NONEXISTBOX	7	A request was made on a box that does not exist.
NOBOXCONN	8	A request was made on a mailbox, but no mailbox connection exists.
BOXINUSE	9	A request has been made with a mailbox that has been checked out by another client.
DELFAIL	10	Failed to deliver a message (the recipient does not exist).
SHUTDOWN	11	The server is shutting down and will be closing all sockets.
COMMTIMEOUT	12	The client's response time has expired; server is going to disconnect.
NOMAIL	13	There is no mail in the mailbox.
INMAIL	14	There is mail in the mailbox.

2.3 Data Restrictions

2.3.1 Identifiers

Identifiers specified by clients must only be those recognizable by the following regular expression: `([^,\.]+\.)*[^,\.]+`

These include expressions of the form: canada.government.primeminister, america.government.president... and so on. It is similar to the *Java package naming hierarchy*[1].

The only disallowed character in an identifier is the comma (0x2C), as it is used to separate identifiers in a list. This leaves support for all international characters, allowing for very localized identifiers such as: 中國.政府.首腦.

[1] http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html

2.3.2 Passwords

All passwords must be 16-byte (128-bit) MD5-hashes. This is the only hashing type that the server supports.

2.4 Variable Length Data Transmission

Some of the data transferred between the server and the client is ambiguous in length (i.e. mailbox identifiers, messages. etc). Two different schemes are used for handling this.

2.4.1 String data

For string data (i.e. identifiers), the data is sent in a *modified* UTF-8 format that looks as follows: `<2-bytes><UTF-8 String bytes>`. The initial 2-bytes are a short integer storing the length of the following UTF-8 string. This limits all identifiers to a maximum length of 8,192 characters. In all sequence diagrams, data of this type will be marked with ***S***.

2.4.2 Binary data

For binary data (i.e. message payloads), the data is sent in a similar way to string data as follows: `<4-bytes><UTF-8 Data bytes>`. The initial 4-bytes are an integer storing the length of the binary stream. This limits all message payloads to a length of 4294967296 bits, or 512 megabytes. In all sequence diagrams, data of this type will be marked with ***B***.

Chapter 3

Action Sequences

3.1 Introduction

The section defines the protocol by which a client's language binding *connector* and the server communicate. All the diagrams are presented as *sequence diagrams*[1] to show how data is sent and synchronization between client and server is maintained.

3.2 Creating a mailbox

Clients must *checkout* a mailbox before they can begin sending or receiving messages. For new clients connecting to the server, a mailbox they can connect to may not already exist. The following sequence diagram in figure 3.1 below explains this process.

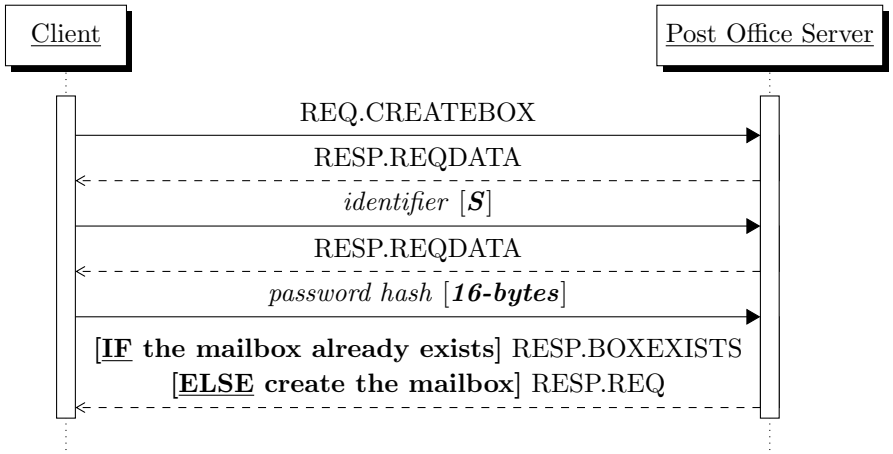

Figure 3.1: Protocol for creating a mailbox.

[1] http://en.wikipedia.org/wiki/Sequence_diagram

3.3 Requesting a mailbox

After a client has created a mailbox, they can *check it out* to begin sending and receiving messages. The following sequence diagram in figure 3.3 below explains this process.

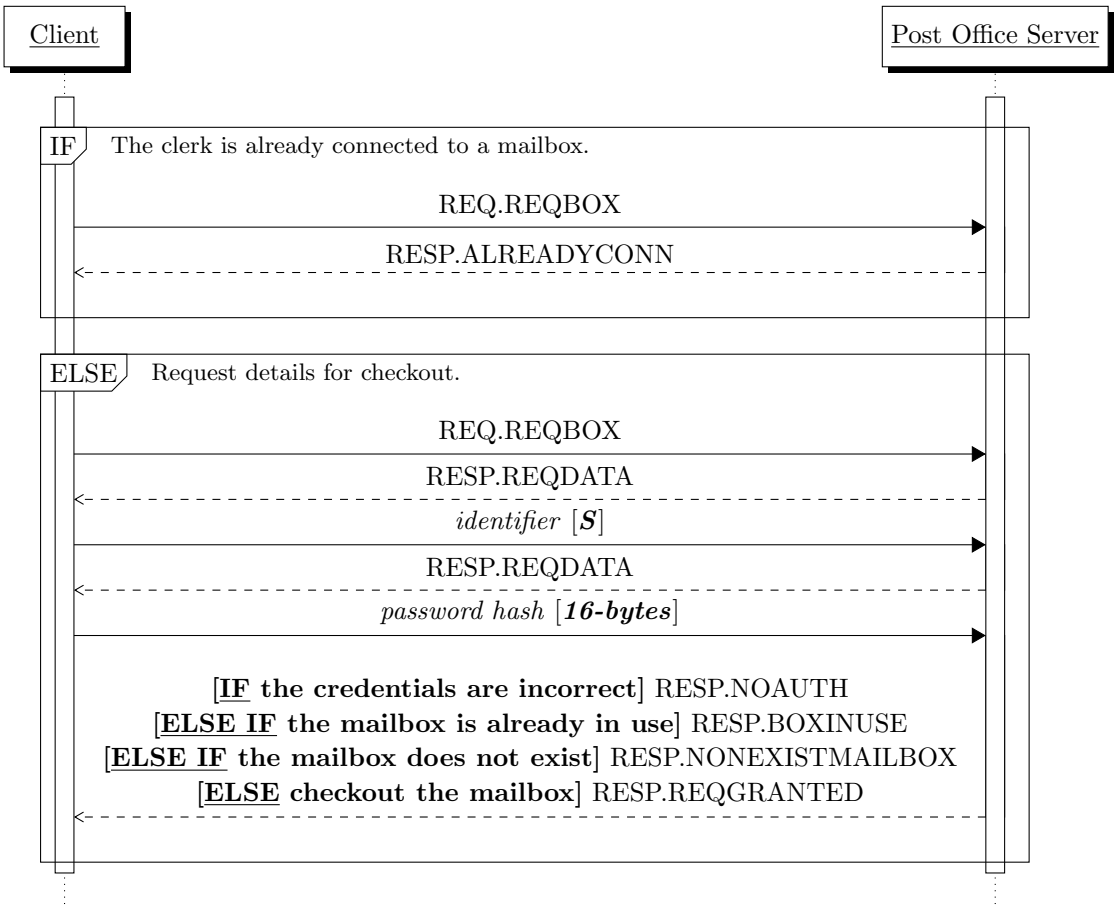

Figure 3.2: Protocol for requesting a mailbox.

3.4 Returning a mailbox

After a client has checked out a mailbox and completed using it, they must *return* the mailbox before checking out another one.

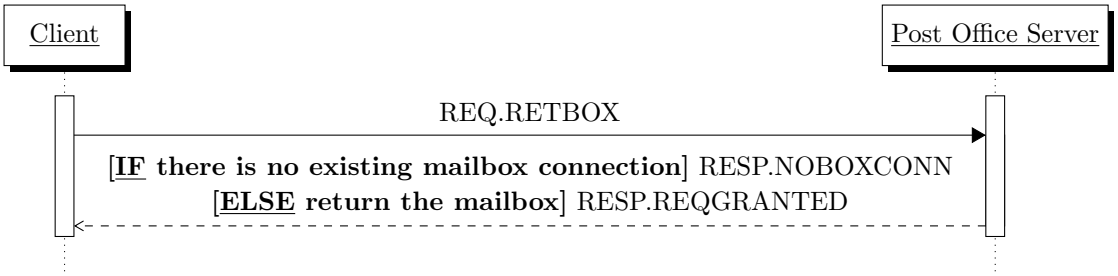

Figure 3.3: Protocol for returning a mailbox.

Keep in mind, a reliable implementation of the *PostOffice* server should automatically return mailboxes should the client randomly disconnect and the socket terminate. It is always good practice to disconnect properly though, rather than causing an exception in the clerk thread on the server.

3.5 Removing a mailbox

In the event of a client disconnecting from the *PostOffice* server, the client may find it desirable to completely remove the mailbox as to not give other sending clients the impression that they are still actively accepting messsages. As long as a client is presently connected to a mailbox, they can destroy it. This process is documented in figure 3.4 below.

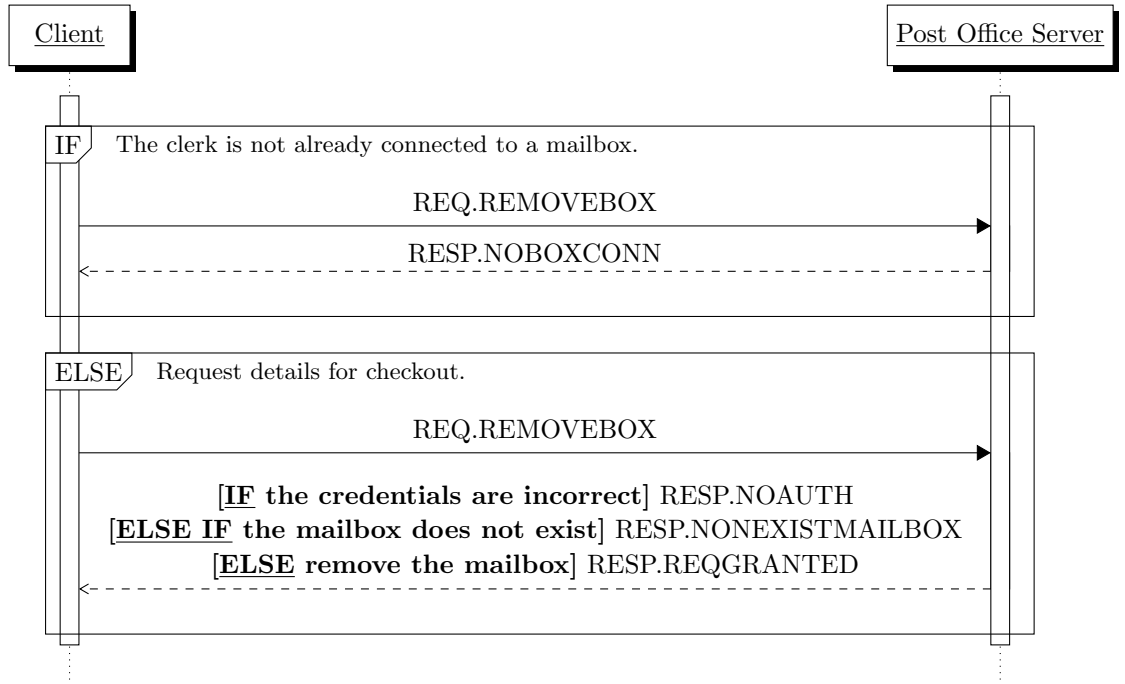

Figure 3.4: Protocol for removing a mailbox.

3.6 Emptying a mailbox

Sometimes a client may find it desirable to completely flush all the messages from the mailbox before waiting on a new one, as to ensure any messages they receive are relatively new. This process is documented in figure 3.5 below.

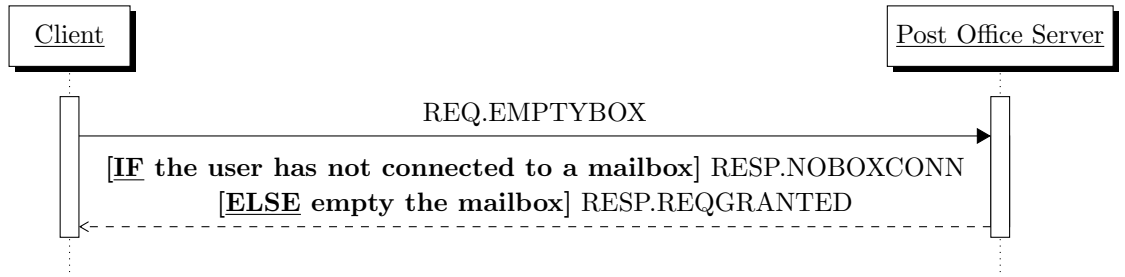

Figure 3.5: Protocol for emptying a mailbox.

3.7 Sending a message

Clients can send a message to an unlimited number of recipients. This process is documented in figure 3.7 below.

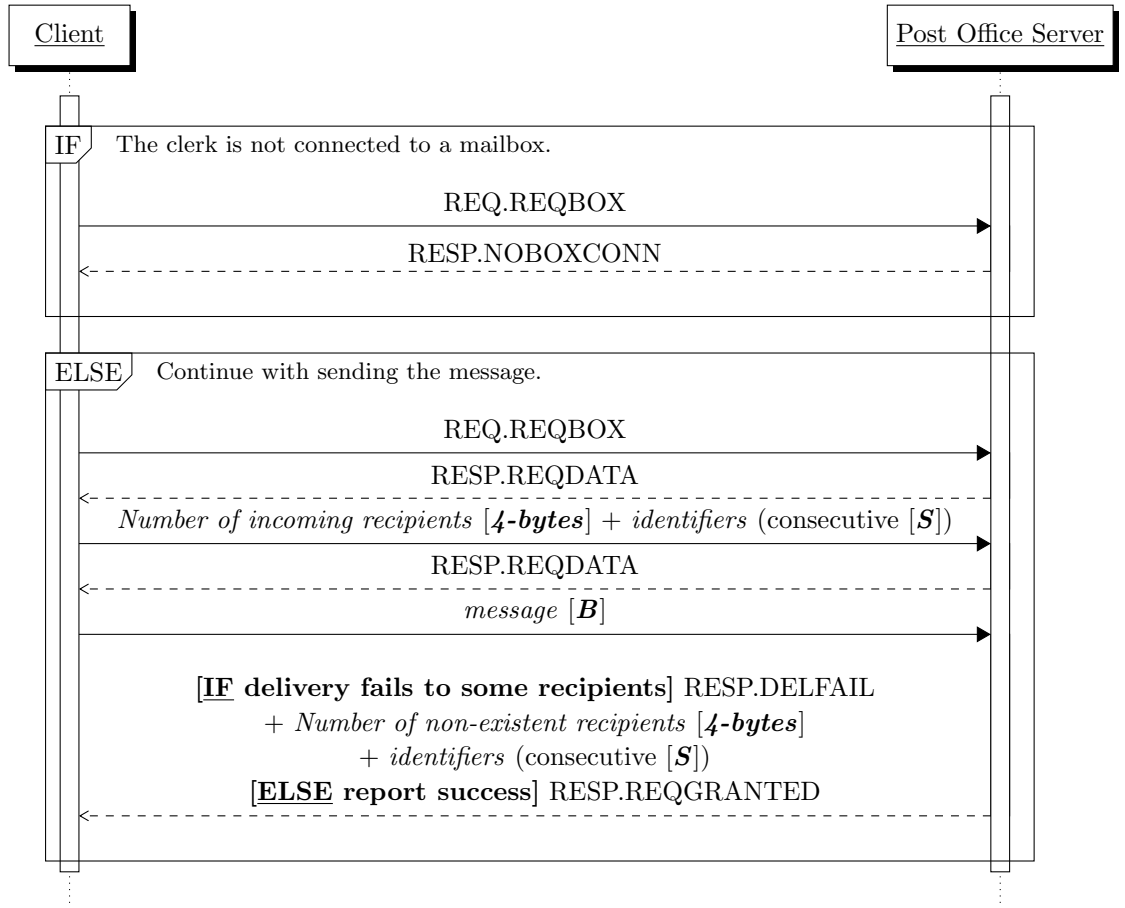

Figure 3.6: Protocol for sending a message.

Recipients are specified first by a 4-byte integer indicating the number of recipients, followed by all the identifiers in the modified UTF-8 format described in section 2.4.1. This format is also used for sending back the list of non-existent recipients to the client, should a delivery failure occur.

3.8 Receiving messages

Clients receive messages in the order they were received in the mailbox. If the client was looking for a specific message, and it is behind several others, then the client must download those messages first before getting to the desired one.

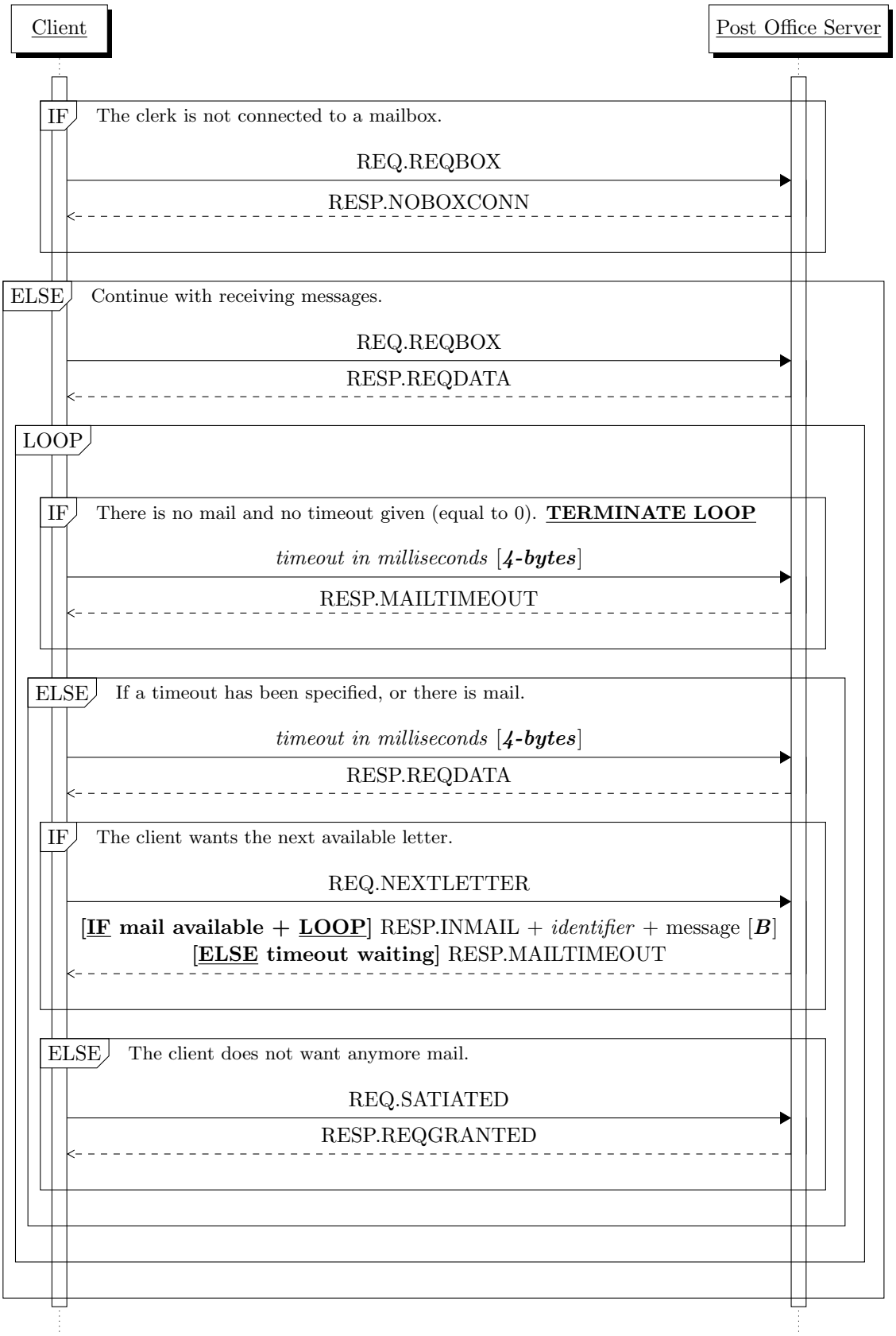

Figure 3.7: Protocol for sending a message.

Receiving mail is the most complicated process the client must perform. When the client requests mail, it

must send the server a "timeout". This is the amount of time the client would like to wait around for a message to show up before relenting to the fact that there is no mail (in milliseconds).

If there is no mail, but the client has specified a timeout, the server will block the client until the timeout has expired. If a message arrives before the timeout expires, the message will be forwarded to the client. If the client is not satisfied with that message, it can respond with *REQ.NEXTLETTER* to continue waiting for more messages until the timer expires. If the client is satisfied, it can respond with *REQ.SATISFIED*. This will tell the server to stop waiting for new mail.

If the client does not specify a timeout, and there is no mail, then the server will immediately notify the client that it has timed out waiting. Otherwise, the same process as above occurs.

If the server times out *while there is mail available*, then it will NOT send the *RESP.MAILTIMEOUT* response. This allows the client to receive as many of the messages in the mailbox as it would like until it is satisfied.

A *good* server implementation should NOT remove messages from the queue until the client has confirmed receipt with either a *REQ.NEXTLETTER* or *REQ.SATISFIED*. This prevents data loss.