A Comparative Evaluation of Imperative and Functional Implementations of the IMAP Protocol

Francesco Cesarini
Erlang Training and Consulting
416 Fruit & Wool Exchange
Brushfield Street, London E1 6EL,
England
francesco@erlang-consulting.com

Viviana Pappalardo
University of Catania
Dept. of Computer and Telecomm.
Engineering
Viale A. Doria, 6
95125 - Catania, Italy
viviana.pappalardo84@alice.it

Corrado Santoro
University of Catania
Dept. of Mathematics and Informatics
Viale A. Doria, 6
95125 - Catania, Italy
santoro@dmi.unict.it

Abstract

This paper describes a comparative analysis of several implementations of the IMAP4 client-side protocol, written in Erlang, C#, Java, Python and Ruby. The aim is basically to understand whether Erlang is able to fit the requirements of such a kind of applications, and also to study some parameters to evaluate the suitability of a language for the development of certain type of programs. We analysed five different libraries, comparing their characteristics through some software metrics: *number of source lines of code*, *memory consumption*, *performances (execution time)* and *functionality of primitives*. We describe pros and cons of each library and we conclude on the suitability of Erlang as a language for the implementation of protocol- and string-intensive TCP/IP-based applications.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

General Terms Algorithms, Performance, Design, Standardization, Languages.

Keywords Erlang, IMAP, Imperative Languages, Functional Languages, Comparative Evaluation

1. Introduction

The choice of programming language and platform to be used in a software system is an issue faced at the start of every project. In the majority of cases, this choice will be influenced by the skills of the programmers and the software development policies of the company. This is quite reasonable from the prospective of the decision makers, but it should not solely rest on these factors alone. This decision should be based on other aspects such as the evaluation of how a language and platform best fits the problem domain. However, even if a particular language could be very appropriate for an application, sensibly reducing the development time while increasing performance, the lack of programmer knowledge results in an exclusion, advantaging other choices better known to the developers.

When several choices of programming languages and platforms are available, debates over the advantages and drawbacks of one paradigm or language or platform with respect to different ones arise, compromising between an objective evaluations of features and personal taste. The issue is that it is not so easy to derive metrics that allow programmers to objectively understand the appropriateness of a language for the development of certain types of applications [13, 15]. When these metrics are available, applying them to a project–either not started or completed–is not an easy task. Best practices often consider an a-posteriori evaluation of an already developed application, aiming at understanding if certain parts of the system (or the whole), which have been particularly hard and time consuming to implement or do not perform could have be realized better using a different technology. Such an evaluation implies an analysis of (*i*) the effort that was needed to develop the application, (*ii*) the difficulties encountered in the development process (due to lack of language constructs or library functions), (*iii*) the complexity of the developed program which leads to a more error-prone application requiring more effort in debugging and testing, (*iv*) the performances of the whole system.

With these aspects in mind, this paper aims at providing a comparative evaluation of different implementations of client libraries for the IMAP email protocol [4]. The starting point was an Erlang project requiring the implementation of an IMAP client. In order to understand the appropriateness of Erlang in similar applications, we performed a series of tests comparing the Erlang solution with some other implementations written in different programming languages, ranging from compiled/imperative, such as Java and C#, to scripting/imperative ones such as Python and Ruby. The aim is to evaluate some parameters like *performances*, *capability to meet user requirements* and *effort* needed to develop the library using that language. This is achieved by both using metrics detailed in the following Sections of the paper and performing a critic analysis of the code and the characteristics of the provided primitives.

The paper is structured as follows. Section 2 provides an overview of related work. Section 3 gives a brief description of the IMAP protocol, showing the basic issues that have to be faced in the development of a client-side library. Section 4 illustrates the basic software architecture of an IMAP client library. Section 5 presents the metrics used to perform the analysis. Section 6 illustrates the implementations we analysed, highlighting their characteristics, and Section 7 summarises the results of the evaluation of them. Section 8 completes the paper with our conclusions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Erlang'08, September 27, 2008, Victoria, BC, Canada.

Copyright © 2008 ACM 978-1-60558-065-4/08/09. . . $5.00.

2. Related Work

Any experienced Erlang developer will confirm that Erlang programs they have written consist of four to ten times less code than their counterparts in C, C++ and Java. This has been an urban legend among the Erlang programmers at Ericsson long before Erlang was released as open source. Almost a decade after its release, however, there still is very little scientific evidence to back up these claims. Examples such as pivot sort using list comprehensions or a distributed remote procedure call server example are used to argue the case. Indeed, they might prove a point when compared to other languages, but when looking at Erlang, there is a necessity to benchmark whole systems, not code snippets.

Similar arguments should be made when looking at performance. As a result, existing language shootouts of sequential code, though valid, do not provide the whole picture. This is enforced by Bogdan Hausman's just in time C compiler [9], which was part of the early versions of the beam emulator, and through the compilation of Erlang to native code in the High Performance Erlang project [11]. In both research projects, a notable increase in the execution of sequential code was achieved. But when complete production systems were benchmarked, the results were more modest. Erlang systems consist of more than the sequential computation of Fibonacci or factorial sequences. They are complex, distributed massively concurrent systems.

In 1989, an Erlang prototype named ACS/Dunder implemented 10% of the features of the MD110 private automatic branch exchange (PABX). Its purpose was to validate Erlang as a language for programming the next generation of telecom applications, providing the productivity comparisons between Erlang and PLEX, the proprietary Ericsson language originally used to develop the MD110. Though the results of the ACS/Dunder project were never made public, Joe Armstrong in 2007 revealed that, depending on the implemented feature, an improvement in design efficiency of a factor three to twenty-two [3] was achieved. These figures were hotly debated at the time, and depending on whether you believed in Erlang or not, were considered highly controversial. As a result of this controversy, the final results were downgraded from an average of eight to three. Quoting Joe Armstrong, "The factor three was totally arbitrary, chosen to be sufficiently high to be impressive and sufficiently low to be believable".

Ulf Wiger, in his 2001 paper "Four-Fold Increase In Productivity and Quality" [18] states that comparisons of projects within Ericsson using C, C++, Erlang, Java and PLEX project yield a similar line per hour productivity and a similar bug density per line of code. What differed between the projects was the final source code volume. Comparisons of C++ projects which were rewritten in Erlang resulted in four to ten times less code, concluding that using Erlang equated to a four to ten-fold increase in productivity and quality. Wiger, however, states that though these numbers would not hold up to a scientific scrutiny, they provide a consistent pattern with experiences of non-Ericsson projects.

The first study to provide scientific evidence and back up the findings by the ACS/Dunder project and Wiger was the comparison of C++ and Erlang for Telecoms Software. Run as a collaboration between Herriot Watt University and Motorola Labs, the research project consisted in refactoring two C++ applications which were in production at Motorola to Erlang [12]. Comparisons were made on the Performance, Robustness, Productivity and impact on programming language constructs. The conclusions were a 70 – 85% reduction in code for the Erlang based system. The code reduction was explained by the fact that 27% of the C++ code consisted of defensive programming, 11% of memory management and 23% of high level communication, all features which in Erlang are part of the semantics of the language or implemented in the OTP libraries. One of the applications completely rewritten in Erlang resulted in a 300% increase in throughput, but that can be argued was a result of Erlang and its light weight concurrency model being the right tool for the job. The Dispatch application in question had lots of concurrency, short messages and little in terms of heavy processing and number crunching. The C++ version had been written with resilience in mind, not performance. Resilience comes almost for free in Erlang. Other conclusions from the project were that robustness and scalability were higher in Erlang while maintenance costs were lower.

3. Overview of the IMAP Protocol

IMAP stands for Internet Message Access Protocol. It allows client programs to access and handle electronic mails stored on remote mail servers. It was proposed by Mark Crispin as an alternative to the POP3 protocol. The current version, IMAP4 revision 1 (IMAP4rev1), is defined in RFC3501 [4]. It supports communication either over a TCP/IP connection using port 143, or over a SSL connection using port 993. The most important advantage of IMAP4 over POP3 is that emails can be stored on the server, allowing any client to access them from any location.

An IMAP4rev1 session is based on reliable client/server network connection over which a request-reply model is run. Client/server interactions start with an initial greeting message from the server and consist of a client request, followed by optional data sent by the server[1] and terminated by a result response. Client and server transmit strings terminated by `CR+LF` character sequence. Each request sent by a client consists of a *Tag*, a unique alphanumeric prefix for each message used to match client request to server reply, and a *Command*. If the command from the client does not require the server to send additional data[2], the server replies immediately with a *tagged* response message, including an indication of the outcome of the interaction (e.g. success or failure). An example of a client/server interaction (a LOGIN command) showing the tagged response is reported below:

```
Client: A001 LOGIN username password
Server: A001 OK LOGIN Ok
```

When data needs to be sent before the response message, the server reply can be split into one or more lines according to the size of the messages themselves. These responses are called *untagged*, they are prefixed not by the *tag* but with characters '*' or '+'. Below is an example of untagged responses returned as the result of the SELECT command:

```
Client: A002 SELECT INBOX
Server: * FLAGS (Junk NonJunk ...
Server: * OK [PERMANENTFLAGS (Junk NonJunk ...
Server: * 4270 EXISTS
Server: * 0 RECENT
Server: * OK [UIDVALIDITY 1186814135] Ok
Server: * OK [MYRIGHTS "acdilrsw"] ACL
Server: A002 OK [READ-WRITE] Ok
```

A client/server session is represented by the finite-state machine depicted in Figure 1. Most commands are available only in certain states, so if a command is sent by the client when in an inappropriate state, an error message is returned.

The initial state, "*server greeting*", is reached after connection. In this state, the server sends a message containing a welcome text and, in some cases, the server's capabilities.

[1] It depends on the command to be executed.

[2] This happens, for example, in LOGIN and LOGOUT commands.

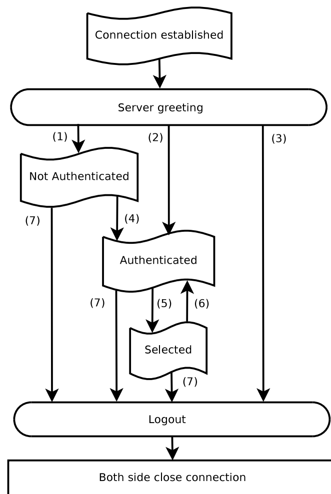

Figure 1. IMAP4rev1 State Diagram

Following this message, the client enters the *Not Authenticated* state (transition 1) unless the connection is pre-authenticated.[3]. In the *Not Authenticated* state, most commands are rejected, because the client has to supply its credentials in order to start its session. This is performed by a proper LOGIN command, if successful (transition 4), the client reaches the *Authenticated* state and must use the SELECT command to choose which mailbox to access. This command contains the name of the mailbox to be manipulated, allowing the client to enter the *selected* state (transition 5). When activities are over, the LOGOUT command terminates the session (transition 7) and closes the connection. In general the *logout* state is entered following the LOGOUT command from the client; however, if a protocol error is detected during the session or a session timeout elapses, the server is allowed to unilaterally trigger the logout and close the connection.

A basic IMAP4 session is characterized by the initial sequence of commands LOGIN and SELECT, followed by commands for manipulating the messages and the mailboxes such as (but not limited to) FETCH, LIST, STORE, COPY and finally terminated by a LOGOUT. FETCH is the important command of the IMAP4 protocol, as it is used to retrieve data items such as header fields, text or attachments of one or more messages. The basic syntax requires the inclusion of additional parameters, expressed by means of proper keywords and used to select specific portions of the message(s). As instance, keyword BODYSTRUCTURE (or simply BODY) allows a client to get information on the structure of a message and its various components. Possible component information might cover attachments, text, their size, encoding character set and the MIME type, to mention but a few. The keyword BODY[*section*] retrieves one or more of the specified sections of the whole body; sections can be HEADER, HEADER.FIELDS, MIME, TEXT, etc.; keyword ENVELOPE returns the structure of the envelope of an email message containing the fields date, subject, from, sender, reply-to, to, cc, bcc, in-reply-to, and message-id.

Together with the data to be retrieved, the FETCH command requires an identifier of the involved message(s). To this aim, IMAP4rev1 defines two type of numeric message identifiers: the *message number* (or *message sequence number*), reflects the position of the message in the given mailbox, while the *unique identifier (UID)* is a number assigned to the message when it is placed in the mailbox (delivered from a sender or as a result of the COPY command). There is a basic difference between these identifiers: while UID is *unique* and its value does not change during the life span of the message, the sequence number can vary by modifying the position of the message in the mailbox; this situation can happen when messages are deleted or inserted.

The response to a FETCH command contains data organized as LISP-like nested lists and enclosed between round parentheses, "(" and ")". At a first glance, this response has a simple structure, but it can get complex when nesting occurs as a result of the message forwarding or the inclusion of an RFC822 form [17]. The structure of a message contains a header, a body section and one or more attachments, unless it is a MIME-IMB [6] message consisting of several sections.

The following example[4] of the FETCH BODYSTRUCTURE is the typical nested structure of the FETCH reply message, where each level of nesting is indicated by a pair of parentheses. The structure of the message reported in this reply is composed of two parts; the text of the mail, written in both plain ASCII and HTML, and an attached binary file named "es.rar", encoded in BASE64. The "boundary" element is a delimiter between the text and the attachment.

```
* 3421 FETCH (BODYSTRUCTURE
 ((
   ("TEXT" "PLAIN" ("charset" "iso-8859-1")
    NIL NIL "quoted-printable" 22 2)
   ("TEXT" "HTML" ("charset" "iso-8859-1")
    NIL NIL "quoted-printable" 393 19)
   "ALTERNATIVE"
   ("boundary"
    "- - _=_NextPart_002_01C72136.F960F3A9")
  )
  ("APPLICATION" "OCTET-STREAM" ("name" "es.rar")
   NIL "es.rar" "base64" 261814 NIL
   ("attachment" ("filename" "es.rar"))
  ) "MIXED"
 ("boundary"
  "- -_=_NextPart_001_01C72136.F960F3A9")
 )
)
```

When the reply contains the headers or the text of an email, complexity increases as the nested structure can encapsulate bare, unformatted ASCII or binary data such as email headers, text, attachments, and other included emails. In this case, the presence of the octets stream is indicated by enclosing its size in curly brackets.

Another commonly used IMAP command is LIST, allowing a client to look into a mailbox hierarchy displaying all folders, relationships and attributes. The syntax of the command requires the *reference* and *mailbox_name* parameters. As mailboxes are hierarchically organized, the *reference* denotes where in the hierarchy the inspection should be started. The *mailbox_name* denotes which mailbox to search in. Its notation allows wild cards, denoted by the special characters '*' or '%'. The reply to the LIST command returns a set of untagged responses, one for each folder of the hierarchy. They contain the folder name, its path, the hierarchy delimiter and folder attributes[5].

[3] Pre-authenticated conditions (transition 2) are indicated by a proper capability in the greeting message. Pre-authentication is in general performed by a check on the client's IP address or other peculiar mechanisms.

[4] Line breaks and indentations have been added by authors in order to make the example clearer; real server's reply is not formatted as shown, even if it is contained in several lines.

[5] According to [4], attributes are chosen among the following values: `\Noinferiors`, `\Noselect`, `\Marked`, `\Unmarked`, `\HasNoChildren`, `\HasChildren`.

3.1 Implementation Issues

As the reader can observe from the explanation of the IMAP protocol, the client/server interactions and the data transport are straight forward to implement, as data exchange occurs by encoding messages in ASCII strings terminated by the CRLF character and sending/receiving them through a socket. The problem with IMAP is in the application layer, as the grammar of the server response is very articulated and complex, putting a requirement on the client to be able to parse and interpret all replies. As a result, most of the implementation effort of an IMAP client is in the parsing of the IMAP server responses. In particular, as has been illustrated above, parsing the FETCH response can be challenging because of the variety of information included in the structure of the message and included attachments. The response to a "FETCH BODYSTRUCTURE" command is one of the most complex to parse, as it can contain many levels of nesting. The response to a "FETCH BODY[section]" command can also present some difficulties in the transcoding activity, as the client has to translate text or attachments contents from the reply encoding (such as BASE64) to an encoding handled by a client program (such as UTF-8 or binary). Finally, as the grammar described in standards [4, 6] also permits user-defined fields or attribute values, another implementation difficulty is making the parser module generic and flexible so as to cater for these situations.

4. Architecture of an IMAP Client Library

This Section gives a brief overview of the software architecture of an IMAP client. According to the specifications provided in the standard [4], we can consider an IMAP client library as composed of the following software layers:

1. **Communication layer**, handling socket connectivity.
2. **Low-level IMAP protocol handler** which is responsible to manage request/reply communication properly adding the *tag* to the request and differentiating *tagged* and *untagged* responses.
3. **IMAP interpreter**, whose task is to parse server replies, transforming them into native types of the target language used in the client implementation.
4. **IMAP FSM**, which handles the finite-state machine of the IMAP protocol (see Figure 1), managing state transitions and ensuring that sent commands are valid in the current state.
5. **IMAP high-level interface**, which implements the various commands of the IMAP protocol by directly using the services provided by the low-level IMAP protocol handler and the IMAP interpreter.

This architecture depicted in Figure 2 is generic and consistent with all of the client implementations we analyzed. According to the specific functionality of the library, however, one or more of the layers could have been omitted depending on the functionality of the client. As an example, a library which does not parse server replies, returning them directly to the user will not have an *IMAP Interpreter* layer; similarly, if the handling of the FSM protocol is not support by the client side, the *IMAP FSM* layer will be not present.

As will be argued in Section 7.1, the presence (and lack) of layers will impact the software writing process of the IMAP client and influence the estimation of the effort involved. Indeed, we can consider that the lower layers such as the *Communication*, *Low-level IMAP protocol handler*, *IMAP interpreter* and *IMAP FSM* (when present) have to be developed first, requiring a big initial effort, while implementing the various commands can be considered quite simple; they can be done incrementally, one at a time, directly exploiting the services provided by the lower layers.

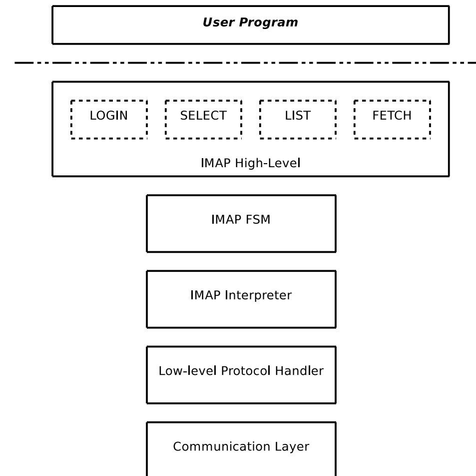

Figure 2. Software Architecture of an IMAP Client Library

```
...
def fetch(set, attr)
  return fetch_internal("FETCH", set, attr)
end

def fetch_internal(cmd, set, attr)
  if attr.instance_of?(String)
    attr = RawData.new(attr)
  end
  synchronize do
    @responses.delete("FETCH")
    send_command(cmd, MessageSet.new(set), attr)
    return @responses.delete("FETCH")
  end
end

def store(set, attr, flags)
  return store_internal("STORE", set, attr, flags)
end

def store_internal(cmd, set, attr, flags)
  if attr.instance_of?(String)
     attr = RawData.new(attr)
  end
  synchronize do
    @responses.delete("FETCH")
    send_command(cmd, MessageSet.new(set), attr, flags)
    return @responses.delete("FETCH")
  end
end
...
```

Figure 3. Code Snippet of the Ruby IMAP Library

The listing in Figure 3, which is the code for the FETCH and STORE commands of the Ruby IMAP library clarifies this aspect. Adding another command involves replicating the basic structure of the code and replacing the command string to handle the new functionality.

5. Evaluation Criteria

As argued in Section 1, providing objective parameters to measure software quality is not an easy task. The literature reports some indicators [5, 12] and, even if some developers and researchers still do not agree on their objectiveness, they should be considered adequate and provide a good degree of confidence [15]. According to this literature, we selected the following parameters:

- *Number of source lines of code* (SLOC);
- *Functionality of primitives*
- *Amount of memory required*
- *Execution time/throughput*

5.1 Source Lines of Code

This is obtained by counting the numbers of lines of all the source files composing the library and removing blank lines and comments. This parameter should provide an indication of the *effort* required to develop the application: roughly speaking, the more the SLOC the more the time required to write the software and to test it. However, as argued in [10], SLOC is dependent of at least three factors: *programmer's skills*, *programming language* and *program functionality*.

Indeed, it is quite simple to understand that sources written by different programmers, with different programming abilities, could sensibly differ in the number of SLOC; in general, if all programs exhibit the same functionalities, the more skilled the programmer the less the number of lines written. In any case, given that most of the implementations compared in this paper are part of the library of the language we tested, we can assume that they have been written by programmers with a very good knowledge of the language itself and therefore they can be considered "good software"; so, as for this point of view, it makes sense to compare the number of SLOC of these implementations.

Another factor that affects the number of SLOC is the programming language employed. Obviously, the presence of certain language constructs has a strong impact on the lines of program used for a certain functionality. For example, extracting the first element of a string is obtained in Erlang by means of a simple one-line statement: `[H|T] = String.` Moreover, if this extraction is performed in the declaration of a function clause, as is often the case, the code becomes even more compact, as the line will contain the function declaration, check a condition (that the string is not empty) and perform a double assignment. In Python or Java, the extraction of the first element requires two lines of code, and additional lines when we intend to also include the "string not empty" check:

Python	Java
`H = String[0]` `T = String[1:]`	`H = string.charAt(0);` `T = string.substring(1);`

`if String != "" :`
`  H = String[0]`
`  T = String[1:]`
	`if (string.length() != 0) {`
	`  H = string.charAt(0);`
	`  T = string.substring(1);`
	`}`

This dependence of SLOC upon the language used is not only fundamental for our study, but also quite desirable as we want to understand the effort needed to develop the IMAP client using different languages. The presence of certain constructs that require less lines of code implies less time needed to write the software and to test it.

Another important aspect that influences the number of SLOC is the functionality the software exhibits; indeed two different implementations of an application which at first glance might appear the same but in reality differ in functionality, cannot be compared in terms of SLOC, as the feature rich implementation will presumably have a higher number of SLOC. This is the reason why, in evaluating the software, the SLOC parameter is weighted by means of the so-called *Function Point Analysis* (FPA) [8]; this is a technique that allows developers to analyze a software implementation and derive a parameter, called *number of function points*, providing an objective evaluation of the functionalities of the analyzed software. Given this measure, weighting the number of SLOC with respect to the number of function points should give a more precise estimate of the required development effort required.

However, performing FPA is not an easy task, above all when dealing with a library. According to [7], FPA seems more appropriate for a complete application or software system rather than a single component. For this reason, in the analysis of SLOC, we used a different approach with the aim of obtaining the same objective evaluation. Indeed, as argued in Section 4, by analyzing the implementations given with their source code (i.e. Python, Ruby, C# and Erlang), we noted their software architectures were very similar to one another. In particular, once the blocks for the low-level protocol, request assembly and reply parsing had been developed, adding a new functionality (i.e. support for another IMAP command) only involved writing a function (or method) that used the primitives provided by these basic blocks. From such a characteristic, we can derive that comparing the number of SLOC of two implementations, even if they support different sets of commands, can be performed by simply analyzing the parts of the software which provide the *same* functionalities, namely the low-level protocol, request assembly, reply parsing and the common commands in the different implementations.

5.2 Functionality of primitives

As argued before, Function Point Analysis is not easy to perform and also not appropriate for a library. On the other hand, in a comparative analysis as the one described in this paper, an understanding of the specific functionalities of a given library should be mandatory. The aim should be to try to understand the *quality* of an implementation in terms of its capability to provide a complete, transparent and flexible support of the IMAP protocol.

Checking the completeness of a solution is quite simple, since it implies to analyse if the library is able to support all or a subset of the commands and functions of the IMAP4 standard. As for transparency and flexibility, an analysis of the software architecture of the solution and the interface of the various primitives (i.e. signatures of function or methods) is needed, in order to understand if a good level of abstraction is provided. Indeed, many commands of the IMAP4 standard require additional parameters which should be passed to the function (or method) implementing that functionality. Even if they are then sent as string in the protocol messages, these parameters may vary in type and semantics, therefore a well-written library should treat them according to their real meaning. As an example, a function implementing the FETCH command could require the additional parameters to be passed as either different arguments or a complete string. In the latter case, there is a lack of a proper abstraction level, since the programmer has to manually create the string to be passed: in other words, to use the library the programmer has to possess a certain knowledge of protocol messages, an aspect which is in contrast to the common rules of software engineering that instead require library/software modules to *hide* specific low-level (protocol) details by providing a high-level flexible and uniform interface.

A similar argumentation can be given for function/method response values, which should reflect the outcome and the reply of the command implemented. If a reply is already interpreted by the library and provided as a structured data type of the language (primitive or derived), the programmer can directly use it without writing additional parsing code. Once again, this is an indicator of a proper encapsulation and abstraction level of the library, characteristics not featured by e.g. an implementation returning raw protocol replies, since, in this case, an additional programming effort is required to properly understand and use them.

5.3 Memory Consumption

Memory consumption is a performance parameter that expresses the memory usage of an application using the IMAP library belonging to a certain language. We developed a simple test program that logs in and fetches a bunch of messages. During the execution of the test program on a Linux OS, we collected information about memory usage by checking the status contained in the "`/proc`" file system, in particular, by looking at *total program size*, *code size* and *data size*.

5.4 Execution time/throughput

The second performance parameter we evaluated was the execution time of certain IMAP commands. As reported in [4], most of the commands imply the execution of a specific activity and reply with a simple success/fail response. Replies from commands such as SELECT, FETCH or LIST, however, are more articulated and probably require complex parsing of the result. For this reason, in order to evaluate the interaction throughput featured by each specific command of the various implementations, we let the client perform both simple and complex commands. Simple commands provide an indication of the performances of the low-level request/reply protocol, while complex commands can be used to measure the performances of the parser when it is present in the tested implementation. In detail, the commands we used for performance evaluation are LOGIN, SELECT, FETCH and LIST.

6. Evaluated Solutions

In this Section the IMAP libraries which constitute part of this study will be described. All of them have been found in the Internet. Some of them are free while others are commercial products. Furthermore, among the free libraries, some are provided with the source code while other are available only in compiled format, so analysis based on SLOC, described in Section 5.1 will not be applicable. The aim of all of the solutions studied is to provide IMAP4 client support to developers. Each library provides only some of the features described in the IMAP standard [4], but they all include the most important commands. These libraries represents different approaches to the solution of the IMAP4 interfacing problem, therefore, to analyze and study their characteristics provides their respective advantages and disadvantages. Each library supports developers in different ways, with varying functionality, different parameters and return values. The following subsections will provide an in-depth description of the characteristic of each of analyzed solutions.

6.1 Java

Even if there are many IMAP solutions for Java, we selected the JavaMail package [2], as it is the library officially released by Sun Microsystems. This package provides a platform-independent and multiprotocol framework which can be used to implement email and messaging applications.

The philosophy behind JavaMail is interesting, as it allows client programs to download message(s) from the remote IMAP server and cache them locally, bringing it closer to POP3 than the other IMAP4 approaches. This package is multiprotocol, in the sense that it allows a transparent access to both IMAP and POP3 servers. A client program using this library must principally, create a `Session` object, useful to retrieve the proper `Store`; the latter determines the correct connection mechanism between client and server depending on the specific protocol required (IMAP, POP3, SMTP). Indeed the `Store` object models a message store and its access protocol. The `Store` object has a `connect` method to establish the connection to the server, and by means of the IMAP "LOGIN" command, supports a simple authentication mechanism.

Once successfully authenticated, the client can access a mailbox by obtaining a `Folder` object from the `Store`. By means of the `Folder` object, the client can open a mailbox in read/write or read-only mode and retrieve data invoking the `getMessages` method. This method returns a vector of `Message` objects which model an email message and provide a set of "`getXXX`" methods[6] to retrieve message data, such as text, flags, sender's and recipient's addresses, etc. In detail, from the point of view of the socket streaming, each time a "`getXXX`" method is invoked, a proper interaction with the server is started to obtain the desired data item; such an information is then cached in the `Message` object in order to make it readily available if it is further needed. Such a data retrieving policy could be considered a drawback, as for performances, since it causes a new protocol interaction with server each time a not yet retrieved message attribute is needed. Indeed, this drawback can be overcome by using a "fetch profile", i.e. the programmer can personalize data recovering by preparing a `FetchProfile` object and then use the `Folder.fetch` method; the latter retrieves, using a single transaction for each message, all data items requested, making them available without needing any further interaction with the server.

[6] For example `getFrom`, `getFlags`, `getAllRecipients`, `getReplyTo`, etc.

The following code is a brief example of what a Java program using the JavaMail library program has to do to retrieve email messages.

```
import javax.mail.*;
...
// Get a Session object
Session session =
    Session.getInstance(properties, null);
// Get a Store object
Store store = session.getStore(protocol);
store.connect(host, port, user, password);
// Open the Folder
Folder folder = store.getDefaultFolder();
folder = folder.getFolder(mailbox_name);
// try to open the folder in read/write mode
folder.open(Folder.READ_WRITE);
Message [] msgs = folder.getMessages();
// Create a fetch profile
FetchProfile fp = new FetchProfile();
fp.add(FetchProfile.Item.ENVELOPE);
fp.add(FetchProfile.Item.FLAGS);
// fetch messages using the profile
folder.fetch(msgs, fp);
...
```

Even if the policy of JavaMail is not the proper way to fetch messages with respect to IMAP4 protocol, its performances are good enough as reported in Section 7.

6.2 C#

The C# solution we analyzed is the "ImapLibrary", available at `http://www.codeproject.com/KB/IP/imaplibrary.aspx` and provided with the source code. This library exploits the dotNet platform, using the framework's native packages such as the socket library and the XML data handling packages. This solution has peculiar return values in some object's methods; in some cases, the output of the parsing activity is identified by an XML file, where each node of the XML tree represents a portion or a section of an email message. The library contains three main source files:

- *ImapBase.cs*; it defines the base class to handle low-level client/server communication, data transmission and reception to/from the socket channel.
- *Imap.cs*; it implements the IMAP4 protocol; it is in charge of preparing client queries, sending the equivalent IMAP4 command and parsing server's response. The `Imap` class defined in this source file is derived from the `ImapBase` class.
- *ImapException.cs*; it defines some exceptions to manage internal errors and protocol faults.

The basic entity of the library is the `Imap` class, which defines a set of methods, each executing a specific command of the IMAP

protocol. In detail, the Login method allows a client to perform both a connection with the server and user authentication. This method verifies if the server is connected and if the client is already logged in. If so, it does not send another login command and choosing the correct policy to manage this condition. After a login, the client is requested to select a mailbox through the SelectFolder method, requiring a string as a mailbox name. Subsequently, to obtain message data, the client can use the following three methods:

- FetchPartHeader. It retrieves the header of the email message or an encapsulated part (i.e. an attachment) and, in particular, implements the IMAP commands "FETCH BODY [HEADER]" and "FETCH BODY [section.MIME]". It requires the UID of the message to be retrieved, the section, and returns (by reference) an ArrayList object; each element of this object is a string containing one of the lines of the header of the retrieved message.
- FetchPartBody. It retrieves the body (i.e. text or attachment data) of a part of a message by using the IMAP command "BODY[section]". The programmer has to specify the UID of the message and the section number; the return value is a string which contains the requested body data.
- FetchMessage. This method retrieves all the parts of a message and produces, as output, an XML file, properly formatted, contained all message data. The method requires the UID of the message and an XmlWriter object to be used for output writing.

The code below illustrates a snippet of a simple client program that uses the Imap library.

```
...
Imap oImap = new Imap();
oImap.Login(host, user, password);
oImap.SelectFolder(mailbox_name);
...
XmlTextWriter oXmlWriter =
              new XmlTextWriter(sFileName,
              System.Text.Encoding.UTF8);
oXmlWriter.Formatting = Formatting.Indented;
oXmlWriter.WriteStartDocument(true);
oXmlWriter.WriteStartElement("Message");
oXmlWriter.WriteAttributeString("UID", sUid);
oImap.FetchMessage(1234, oXmlWriter, true);
oXmlWriter.WriteEndElement();
oXmlWriter.WriteEndDocument();
...
```

As the description and the code above highlight, the library performs a partial parsing of the reply as for the methods FetchPartHeader and FetchPartBody, and a full parsing in the method FetchMessage, but, in the latter method, the output is not directly usable by a client program: instead, the programmer is requested to re-interpreter the XML file in order to obtain the information s/he needs.

6.3 Python

Python [16] offers an IMAP client module delivered with its standard library called "imaplib". As with all modules of the Python library, imaplib is free and provided with its source code.

The basic component of the library is the IMAP4 class, which defines methods for all the commands of the IMAP standard, providing a 1:1 mapping. All of the library methods have similar interfaces; all of them require a string which represents the parameter(s) for the command to be sent to the server, therefore the library creates the IMAP command by simply appending the string received as argument and adding the *Tag*. Similarly, the reply is returned by the method "as is", without any parsing, which if required, must be done by the client. Indeed, the return value of all methods is a tuple with two elements: the first element is the *tagged* response, representing the outcome of the command, while the second element is the *untagged* response containing the data sent back by the server.

An interesting capability of this library is the authentication mechanism, which is provided through the "AUTHENTICATE" command. It does not use clear text, relying on a more flexible and secure authentication model described in [14].

A python client program which wants to connect to an IMAP4 server must create an IMAP4 object and use it to invoke the login or authenticate method. Once the connection is established, the client can select the folder to examine and call the fetch method to pick up message(s) and associated data. This method requires two parameters: the first parameter is the message sequence number, expressed as an integer, or a range of messages, provided as a string in the form ``first message number:last message number''; the second parameter is a string representing the complete argument to be appended to the "FETCH" command. The code below shows a basic client program in python using the imaplib:

```
import imaplib
...
imap = imaplib.IMAP4(host)
result, data = imap.login(username, password)
# The default mailbox is 'inbox'
result, data = imap.select()
# Let's get the text and the UID of the
# 10th message
result, msg_data =
  imap.fetch(10, '(UID BODY[TEXT])')
# Let's get the 'From' and 'Subject' fields of
# the messages from 1 to 5
result, msg_data =
  imap.fetch('1:5',
         '(body[header.fields (from subject)])')
```

The library provides additional features since offers other commands, such as APPEND, CREATE, DELETE, to modify the selected mailbox, or UID, FLAGS, LIST, to retrieve information about message or mailbox characteristics.

6.4 Ruby

Ruby [1], like python, is an interpreted (scripting) language. Its basic library offers an IMAP module called net/imap which comes as part of the language distribution. This library is free and the code is accessible to the developers.

The library provides a class Net::IMAP whose interface is similar to the module provided for the Python language. However, unlike the IMAP Python implementation, the Ruby library performs a parsing activity and returns the server response to the client program by means of proper language types which can be directly used. The code below illustrates a simple ruby client program:

```
require "net/imap.rb"
...
imap = Net::IMAP.new(host)
imap.login(username, password)
imap.select(mailbox_name)
info = imap.fetch(20,
       "BODY[HEADER.FIELDS (FROM SUBJECT)]")
message = imap.fetch(10..15, "BODY[HEADER]")
```

As the example illustrates, the client program must create a Net::IMAP object, specifying the server address as parameter, and then invoke the "login" or "authenticate" method. Subsequently, the client can select a mailbox (select method) and manipulate it

through the proper commands defined in the library. As for message retrieving, the `fetch` method needs two parameters resembling the fetch primitive of the python implementation, even if the reply is properly parsed. Indeed, the `fetch` method returns server's reply as a native Ruby type, i.e. a structured type composed of two fields: `seqno` and `attr`, the former is the message sequence number while the latter is a hash table in which each item is a couple {*key, value*}; here the *key* is one of the arguments specified with the "FETCH" command (e.g. "FLAGS", "BODYSTRUCTURE", "UID", etc.) and *value* is the associated data. This data is generally returned as is, without any parsing, unless one of the following arguments are passed: "BODYSTRUCTURE", "BODY[TEXT]", "BODY[section]", "ENVELOPE". In such cases, the *value* field may be one of the following defined types:

- `Net::IMAP::BodyTypeBasic`, which represents the structure of the body of a simple message;
- `Net::IMAP::BodyTypeMultipart`, which represents a message composed of more than one part, that is a text and one or more attachments;
- `Net::IMAP::BodyTypeText`, that contains the text of a message;
- `Net::IMAP::BodyTypeMessage`, which represents a message of the type MESSAGE/RFC822, i.e. a message encapsulated in another message, as it happens in the case of forwarding an email.

There structured data types are very useful, as, by properly navigating their fields, a programmer can directly access all the information regarding the fetched messages. A proper exception is finally raised if the command does not succeed due to a protocol error or a bad parameter.

6.5 Erlang

The client library we implemented in Erlang as part of this study represents an interface between a front-end and an IMAP4 server. It handles both TCP and SSL client/server connections. The library is based on OTP and works as a stand-alone Erlang application. It manages the most important IMAP4 command such as LOGIN, SELECT, FETCH, LIST, IDLE performing an analysis of server responses in order to create an Erlang representation of email data. Indeed, the library parses server response, verifies it correctness according to IMAP4 standard, and produces an Erlang-native result value.

The library is mainly composed by three modules: `im_client`, the interface between the front-end and the IMAP4 server, it implements all the IMAP4 commands; `scanning,` which performs a lexical analysis of the reply recognizing and carrying out tokens; `parsing`, that instead parses the output of the previous module by generating an Erlang-native term for the server response.

The basic module of the library is `im_client`, which also provides the interface functions to execute IMAP commands. The module is a `gen_server` and the process' status data hold, together with information such as the socket connection, the state of the finite-state machine of the protocol (see Section 6.5) in order to perform correct sending and handling of commands.

The interface provided by `im_client` is simple and quite similar to that of other described libraries. Starting the activities implies to activate the `gen_server` process managing the protocol through the `start_link` function, which takes two parameters, mail server address and connection type ("tcp" or "ssl"). Subsequently, the client can invoke the `login` function to perform authentication; it uses the basic authentication mechanism is based on username and password, and, like all other libraries (excluding the Python implementation) currently does not support the AUTHENTICATE IMAP4 mechanism. The next step is mailbox selection, which is performed through function `select`; after this, messages can be retrieved using the `fetch` function, which requires two parameters, the message or the list of messages to be retrieved and the additional arguments for the FETCH command. A flexible way to specify such arguments has been implemented, indeed messages to retrieve can be specified by providing a list of their message numbers (or UIDs); moreover, if a range of messages is required, it is possible to specify, as an element of the list, a tuple in the form `{first,last}`. As for the arguments of the FETCH command, they are provided as a list of strings, such as `[''flags'', ''bodystructure'', ''size'']`, which are then suitably interpreted by the library in order to prepare the complete FETCH command. Therefore, these parameters are not barely concatenated with "FETCH", as it happens for example in Ruby and Python, but they express the precise information that a programmer wants to obtain from a message; for example, to retrieve the sender, the recipient, the subject and the text of an email, it suffices to pass the term `[{''body'', 1, 1}, ''from'', ''to'', ''subject'']`: the library is able to interpret the provided information, prepare the proper command, which in this case is "(BODY.PEEK[1] BODY[HEADER.FIELDS (FROM TO SUBJECT)])", and suitably parsing the received response.

All of the IMAP protocol functions of the `im_client` module return a tuple containing two elements: `{ResultValue, Parsed Reply}`, the former is an atom representing the server response to the command[7], while the latter is the server's reply data, as provided by the parser modules. In functions `select`, `fetch` and `list`, this second element of the return value is a "proplist"[8] ({*key,data*}), in which the *key* is a symbol with expresses a message data item and *data* is the associated value. As an example, the function call, `im_client:fetch ([5,6], [''bodystructure'', ''date''])`, with generates the IMAP command "FETCH 5:6 (BODYSTRUCTURE BODY[HEADER.FIELDS (DATE)])", replies (on success) the following data:

```
{ok,
  [[{"seq_no",5},
    {"bodystructure","text/plain","7bit","24","834"},
    {"date",{{2007,3,6},{23,2,52}}}],
   [{"seq_no",6},
    {"bodystructure","text/plain","7bit","20","577"},
    {"date",{{2007,3,6},{17,9,0}}}]]}
```

The piece of code below illustrates the use of this library:

```
...
im_client:start_link (Mailserver_name, ssl),
{ok, _} = im_client:login (Username, Password),
{ok, MBoxInfo} = im_client:select ("INBOX"),
{ok, MsgSet1} =
    im_client:fetch( [{1,3}, 10],
                     [{"body", 1, 1}, "from",
                      "to", "subject"]),
{ok, MsgSet2} =
    im_client:fetch( [{15, 20}],
                     ["bodystructure"]),
...
```

The Erlang library also offers some interesting features. One of them is the ability to check the connection and re-instantiate it, if needed; indeed the library avoids some protocol errors such as the "brutal closure connection", which occurs when a login is not performed within a timeout after establishing the connection or when, due to a long period of inactivity, the server unilaterally

[7] It can assume the value 'ok', 'not' or 'bad'.

[8] In particular, for the `fetch` command, it is a list of "proplists".

decides to abort the connection. To this aim, before sending each command, the library verifies if the user is just authenticated and the connection has been set up. Another interesting feature is the ability to support multiple client commands, since their sequencing is then handled internally. Finally, since the library is based on OTP concepts, it can be run as an OTP application embedded in a more complex Erlang system, running in the same memory space as the application using it.

7. Results

This Section compares software productivity measures of all IMAP libraries evaluated in this work. The software metrics used for this comparison are the followings:

- *SLOC*, which measures software size and exactly the number of logical software lines;
- *Functionality of primitives*, which analyses the functionalities of primitives provided by the IMAP libraries;
- *Amount of memory*, which measures the space cost of the library;
- *Execution time*, which measures the time required by different libraries to perform protocol activities.

The following subsections will report all results that have been collect during this work.

7.1 SLOC

The SLOC is a software metric that provides a measure of the effort needed to create an IMAP library in a specific programming language. As discussed in Section 5, libraries analyzed in this paper present different characteristic as they do not provide the same number of functionalities, in terms of both IMAP primitives supported and parsing of server responses. For these reasons, we decided to consider the pieces of code, of the various libraries, which implement (more or less) common IMAP primitives.

Language	*Lines*
Erlang	1,189
Python	472
Ruby	1,612
C#	1,089
Java	n/a

Table 1. SLOC

Table 1 summaries the lines of code calculated; the highest number of SLOC is that of Ruby implementation, while the shortest library is the Python one. Erlang and C# feature a similar number of SLOC, while, for JavaMail, this measure cannot be performed since the source code is not released.

The small SLOC number featured by Python is justified by the complete absence of any parser. With respect to the architecture illustrated in Section 4, the Python library only possesses the *Communication Layer*, the *Low-level Protocol Handler* and the *IMAP High-Level Layer*.

On the other hand, Erlang and Ruby implementations support a *full* parsing of servers replies, and as a result present a higher number of SLOC. The parser in the Ruby library is however more complex than the Erlang one due to the definition of data types and an intensive use of complex regular expressions. The lines devoted in Ruby to reply analysis were 1048 in contrast to the 818 from the Erlang implementation. The reduction in Erlang is a result of making use of function clause matches in order to directly identify reply tokens and perform the appropriate data extraction. This aspect not only improves performances (see Subsection 7.4), but also the comprehension and maintainability of the source code.

Finally, the C# implementation presents a number of SLOC comparable to that of Erlang. This is quite interesting even if C# does not provide full parsing of the server replies. It does not parse the (often complex) result of the SELECT command, something Erlang does in full. It does not support the LIST command and only some parts of the FETCH command, both of which Erlang also fully supports.

7.2 Functionality of primitives

In this Subsection, we will describe the functionalities offered by the various IMAP libraries comparing and contrasting their characteristics. In particular, we will focus on the primitives "LOGIN", "SELECT/EXAMINE", "FETCH" and "LIST". Table 2 summarizes and compares the results of this analysis.

7.2.1 Login

The first functionality we deal with is authentication; to this aim, each library has method or function that performs the IMAP LOGIN command. All of the implementations require the "user_name" and "password" to authenticate client to the server. The Python and Erlang implementations return a result value indicating the success or the failure of the login command; other libraries raise an exception in the case of failed authentication.

7.2.2 Select

The second functionality we analyzed is the selection of the mailbox. The client program can access the desired mailbox by means of the SELECT or EXAMINE command: the former opens the mailbox in read/write mode, the latter in read-only mode. With the exception of JavaMail, all the libraries feature a proper SELECT/EXAMINE function or method. The Java implementation, however, uses a different approach as it provides an `open` method in the `Folder` object which represents the mailbox. This method requires a parameter that indicates how to open the specified mailbox (READ_ONLY or READ_WRITE). Once a folder/mailbox is open, its mode cannot be changed, e.g. from read-only to read-write or vice versa: the `Folder` object must be "closed" and then re-opened with the new mode. This policy is due to the class hierarchy and model used by JavaMail and could make the client more complex: indeed, even if the IMAP4 protocol manages a SELECT command followed by the EXAMINE command (to change the mode to read-only), the Java library does not manage this condition and must close the selected folder and re-open it.

As for the reply to the SELECT/EXAMINE command, C# does not perform any parsing while Java does it, but manages the resulting information internally.

Python returns the total number of messages in the mailbox (which is given by the EXISTS keyword in one of the untagged responses of the server) while Ruby uses a different approach; indeed, the reply to the `select` method is bare text (with no parsing) of the server response[9], but the parsing is internally handled: the `responses` attribute of the `Net::IMAP` object is a dictionary whose keys "EXISTS" and "RECENT" are the number of total and new messages of the mailbox, returned as a result of the SELECT command.

The Erlang library provides the most complete parsing of the SELECT command; the provided `select/1` function returns the tuple {`result_value, parsed_response`}: the former parameter is the outcome of the command (success or failure) and the latter

[9] In particular, the return value is the Ruby structure `Net::IMAP::TaggedResponse`, which contains other two structs, `Net::IMAP::ResponseText` and `Net::IMAP::ResponseCode`.

Command/Primitive	Erlang	Python	Ruby	C#	Java
LOGIN	Yes	Yes	Yes	Yes	Yes
SELECT	Full parsing	No parsing	Partial parsing	No parsing	Internal w/full parsing
FETCH	Parametric Query Flexible expression of ranges Full parsing	Parameters as string Ranges as string No parsing	Parameters as string Range or single messages Partial parsing	Only some queries supported Single messages Partial parsing	Encapsulated Queries Single messages Full parsing
LIST	Full parsing	No parsing	Full parsing	n/a	Full parsing Supported in current context

Table 2. Comparison of Functionality of Primitives

element is a "proplist", i.e. a list of tuples of the form {`keyword, value`}. In this proplist, the keyword is an Erlang atom representing the status information of the mailbox such as `flags`, `exists`, `recent` etc., while the second tuple element is the related value[10].

7.2.3 Fetch

The FETCH command is treated by Python and Ruby libraries in a similar way. As introduced in Sections 6.3 and 6.4, the method provided requires two parameters: a sequence number or a range and the string command to send to the server. The Ruby implementation parses the server response and generates a proper result value by means of Ruby structured type. The Python library, one the other hand, only parses of the tagged response, interpreting the outcome of the command, returning the untagged response "as-is".

The Java library provides a `fetch` method with the `Folder` class that downloads locally remote messages; once fetching is complete, messages are internally cached so new requests stored locally will not result in a query to the server. Even if the cache allows the client to pick up data quickly, it is not able to send a user-defined fetch command using this method. Moreover, it can retrieve only the fetch attributes provided defined in the FetchProfile class.

The C# implementation has three fetch methods that retrieve different section of a message:

- `FetchPartBody`, which retrieves the body text of a message, the text is returned to the client as string;
- `FetchPartHeader`, which picks up the header of a message and returns its fields within an ArrayList;
- `FetchMessage`, which generates an XML file containing the field of header and body structure reply.

Therefore, the type of the reply and the support for parsing depend on the method called; this is quite unusual because according to good software design, methods of the same class with similar functionalities should behave similarly.

The Erlang library interface for the fetch primitive is strongly based on native data types. As detailed in Section 6.5, the programmer can specify both single messages and message ranges within the single query, and the data items to be retrieved by means of proper list of pre-defined symbols. The reply organizes parsed data items in a "proplist", so it is quite easy, for a programmer, to pick the requested item by means of a simple `proplists:get_value/2` function call.

7.2.4 List

The "LIST" command aims at examining the hierarchy of the folder/mailboxes. It is similarly supported by all libraries with the exception of the C# implementation, which does not provide this functionality.

[10] It can be either a single value (for instance the value of the EXISTS untagged response is only a number representing the total number of message in the mailbox), or a list of values if the untagged response contains many value, as the FLAGS untagged response.

The JavaMail package provides a `list` method in the `Folder` class; therefore a client that has opened a mailbox can investigate only its context.

The methods/functions provided by Python, Ruby and Erlang have the same signature of the list function and require two arguments; the *reference name*, which represents the context in which to investigate, and the *mailbox name*, with possible wildcards. The main difference is in the return value. The Python library does not perform any parsing and returns the (untagged) response as a string for further analysis. The Ruby implementation instead parses the response by returning an array of `Net::IMAP::MailboxList` structures whose fields represent the attributes of a folder, returned by the command. The Erlang implementation carries out, as result of a "LIST" query, the tuple {`result value, list`}; the second element is a list containing other lists; each inner list represents a folder that matches the LIST pattern and its items are tuples of the form {`keyword, value`}[11] carrying the attributes associated with this folder. An example of the return value `list` function of the Erlang implementation is reported below:

```
{ok,[[{name,"mailbox_name_A"},
      {separator,"."},
      {noinferiors,false},
      {noselect,false},
      {marked,false},
      {unmarked,false}],
     [{name,"mailbox_name_B"},
      {separator,"."},
      {noinferiors,false},
      {noselect,false},
      {marked,false},
      {unmarked,false}]]}
```

7.2.5 Discussion

The comparative outcome of the analysis of the functionality of primitives is reported in Table 2. The Table highlights that the most featured and transparent implementations are those in Erlang and Java, while the most feature-poor implementation is the Python one. Erlang and Java solutions, however, differ in some aspects of the software architecture because, while the Erlang implementation provides a direct interface to the IMAP commands, JavaMail is based on an object hierarchy with abstractions such as `Folder` and `Message`; from this point of view, JavaMail supports a higher level interface but undoubtedly introduces an overhead.

7.3 Amount of memory required

This metric evaluates the amount of memory required by an application using the various IMAP libraries. The results provide the memory imprint computed for the various IMAP clients at run-time while they are using these libraries. The second column in Table 3 contains the total memory used by the program, the third one represents the memory taken up by the text code and the last one contains

[11] It is a "proplist" again.

the amount of memory used by the stack and heap of the program to keep its data[12].

Language	*Total*	*Code*	*Data/Stack*
Erlang	8,056	2,868	4,656
Python	6,748	4,400	1,684
Ruby	14,942	4,332	10,386
C#	18,800	11,148	2,748
Java	212,852	14,428	190,060

Table 3. Quantity of Memory Space Used (values are in KB)

It should be noted that this measure mainly reflects the usage of the virtual machine and the library, since all the tested languages are interpreted. As reader can deduce, the highest amount of memory is required by the Java client program, which needs 212 MB of main memory; indeed, since the code is only 14 MB large, the most of the space is devoted to the class library which is loaded and compiled (by the JIT) at the startup of the JVM.

The lighter platform, from the point of view of the total size, is Python: the VM size is only 4 MB and also the library is quite small.

Erlang provides very interesting performances: its virtual machine is very small and is the lighter among the tested languages as it occupies only 2 MB. The size of data space (4.5 MB) is due to the Erlang/OTP runtime which is activated at the startup of the system.

7.4 Execution time/throughput

This Subsection compares the execution times of some primitives of the IMAP implementations described in this paper. The aim is to evaluate the performance of each library in order to compare and contrast the throughput. These results will allow us to get a sense of which programming languages and which architecture best meets the productivity requirements and performance of a distributed system. In general, performances are determined by communication and process management, therefore, for each library, we measure the time of the network interaction. Network interaction is the time required to send and receive protocol commands together with the time taken to parse the server result.

All the tests have been conducted using the same mail server and the same set of email messages. The testing platform (the client) is a 2 GHz Intel Centrino PC, equipped with 1 GB of RAM and Ubuntu Linux 7.10 (kernel 2.6.22). Client and server are connected to the same fast-Ethernet LAN. The runtime systems used for the languages tested are reported in Table 4.

Language	*Platform/Release*
Erlang	R11B-5 (erts 5.5.5)
Python	2.5
Ruby	1.8
C#	Mono JIT 1.2.4
Java	Sun's JDK 1.6

Table 4. Platform/Runtime Systems for the Libraries Tested

Table 5 reports the execution times obtained for each command on all the IMAP libraries. Unluckily, the Java and C# libraries do not provide exactly the same functionality offered by other IMAP libraries; indeed, JavaMail and ImapLibrary have a different policy to retrieve data when compared with the other implementations. Furthermore, they present particular primitives which perform a composite IMAP4 command which cannot be mapped to any of the other commands generated by other libraries. For instance, the C# library offers the `fetchMessage` method to send the IMAP4 "FETCH BODYSTRUCTURE" and "FETCH BODY[HEADER]" commands, which produces an XML file as the outcome of its parsing activity. In addition, the high-level interface of the C# implementation does not offer primitives to directly perform a basic IMAP4 command; only JavaMail allows developers to send a user-defined command overriding the `doCommand` method of the `IMAPFolder` class. This last function executes the user-defined IMAP4 command and returns an array of `Response` objects which contain the ASCII server response. As a result, this method does not perform any parsing activity.

As discussed in the previous Section, the JavaMail package offers a large range of classes and methods to perform IMAP4 protocol activity. The retrieval of data and handling of the mailbox are encapsulated within a Java object, hiding the whole mechanism from the developers. Even if this policy simplifies the client program, it made it hard to compute or compare the execution time of some of the primitive, resulting the "not-available" for some measures.

Comparing the execution times reported in Table 5, the reader can notice that the Ruby implementation provides the worst performances and also the C# solution does not present good execution times. At first sight, the Python library seems the best; we should remind that it does not perform any parsing activity, so the reported measures are (more or less) the times of the network/socket communication.

As for the Erlang and Java solutions, as the Table shows, they feature comparable performances for the commands providing the same functionalities (network transaction + parsing), which are reported in columns 1, 2, 3, 4, 7 and 8: with the exception of "FETCH BODY[TEXT]", the Erlang solution executes in less time with respect to the Java library[13]. This is an interesting result, since we should consider that the Java program is executed in native code (it is compiled by the JIT), while, in Erlang, execution is mainly interpreted. This result is important as it confirms the ability of Erlang to fit the requirements of a distributed application, not only in terms of *distribution* and *fault tolerance* (handled by the native mechanisms of OTP), but also for the performance aspects concerned.

8. Conclusions

In this paper, we reported the results of a comparative analysis of five client-side IMAP protocol libraries implemented in different programming languages: Python, Ruby, C#, Java and Erlang. The aim is to evaluate the performances of Erlang in order to get a sense of the ability of this programming languages and IMAP implementation architecture to meet requirements of productivity and performance for a distributed system. We selected different parameters to perform our comparison: number of *Source Lines of Code (SLOC)*, which provides an assessment of effort needed; *functionality of primitives*, which highlights the way in which IMAP protocol is supported and the quality of the interface to be then used by the developer; *amount of memory*, which measures the space cost of the application; *execution time*, which measures the time required to perform certain basic and critical IMAP activities.

From the analysis, we can conclude that the Erlang library can deliver the requirements of functionality and performance for a distributed system. From the latter point of view, if we consider

[12] `Total Size` also includes read-only data and other segments of memory space, this is the reason why this value is greater than the sum of `Code` and `Data/Stack` columns.

[13] The Erlang time in row 6 can be compared to the Java time in row 7 since these two commands are comparable in terms of both network overhead and reply structure.

#	*Command/Primitive*	*Erlang*	*Python*	*Ruby*	*C#*	*Java*
1	LOGIN	28.6	45.7*	26.1	32.7	30.7
2	SELECT	2.5	2.1*	44.3	13.2	4.5
3	FETCH BODY[TEXT]	43.0	41.1*	80.2	40.0	40.2
4	FETCH BODY[HEADER]	2.5	0.2*	44.5	5.3	2.9
5	FETCH BODY[HEADER.FIELDS]	2.3	0.2*	43.0	n/a	1.3*
6	FETCH BODYSTRUCTURE	1.8	0.2*	80.2	n/a	1.2*
7	FETCH (ENVELOPE SIZE BODY.PEEK[HEADER.FIELDS] ...)	n/a	n/a	n/a	n/a	1.9
8	LIST	0.7	0.9*	52.1	n/a	6.3

* = command executed without any parsing of the response

Table 5. Execution Times of Tested Commands (values are in milliseconds)

the implementations with similar functionalities (i.e. structured parsing of the responses), the execution times of the Erlang solution are high and comparable to those of Java, while the worst results are featured by the Ruby implementation. And even if the best results have been calculated for Python, it should be noted that its library lacks in-depth parsing activity and transparency of function signature. These results confirm that Erlang has significant benefits not only for the rapid production of robust distributed system, but also for the achievement of desired performances without high memory cost.

9. Acknowledgments

This work was a collaboration between Erlang Training and Consulting, UK, and the University of Catania, Italy.

References

[1] http://www.ruby-lang.org/en/.

[2] http://java.sun.com/products/javamail/downloads/index.html, 22Oct. 2007.

[3] J. Armstrong. A History of Erlang. In *Proceeding of History Of Programming Languages*, 2007.

[4] M. Crispin. INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. *ARPANET Request for Comment No. 3501*, 2003.

[5] T. DeMarco. Yourdon Press, New York, NY, USA, 1982.

[6] N. Freed and N. Borenstein. MIME (Multipurpose Internet Mail Extensions) Part One: Format of Internet Message Bodies. *ARPANET Request for Comment No. 2045*, 1996.

[7] H. D. Garmus D. *Function Point Analysis. Measurement practise for successful software projects*. ADDISON WESLEY, Nov. 2000.

[8] F. I. F. P. U. Group. *Function Points Counting Practices Manual (version 4.1.1),* `http://www.ifpug.org/`*, WWW*. 2000.

[9] B. Hausman. Turbo Erlang: Approaching The Speed of C. In *Proceedings of the Implementations of Logic Programming Systems Conference*, 1993.

[10] D. Hubbard. The IT Measurement Inversion. *CIO Enterprise Magazine*, 1999.

[11] E. Johansson. HiPE: A High Performance Erlang System. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, 2000.

[12] D. King, H. Nyström, and P. Trinder. Comparing C++ and Erlang for Motorola Telecoms Software. In *Proceedings of the International Erlang User Conference*, 2006.

[13] T. J. McCabe. A Complexity Measure. *IEEE Transaction on Software Engineering*, 2(4), 1976.

[14] J. Myers. Simple Authentication and Security Layer (SASL). *ARPANET Request for Comment No. 2222*, 1997.

[15] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[16] Python Software Foundation. http://www.python.org.

[17] QUALCOMM Incorporated. Internet Message Format. *ARPANET Request for Comment No. 2822*, 2001.

[18] U. Wiger. Four-fold Increase in Productivity and Quality. In *Proceedings of the FEmSYS, Deployment on Distributed Architectures*, 2001.