A Comparative Evaluation of

Imperative and Functional Implementations of the IMAP Protocol

ACM SIGPLAN Erlang Workshop
Victoria (BC), Canada
September 27th, 2008

Francesco Cesarini,
Viviana Pappalardo
Erlang Training & Consulting

Corrado Santoro
University of Catania

Agenda

- Purpose
- IMAP Protocol Overview
- Evaluation Criteria
- Evaluated Solutions
- Result and Conclusions
**Purpose**

Comparative analysis of several IMAP implementations evaluating
- Performance
- Capability to meet user requirements
- Effort needed to develop the library

We use the metrics
- Source Lines Of Code
- Functionality of primitives
- Memory Usage
- Execution Time

**IMAP Protocol Overview (1/2)**

Client-Server protocol communication
- Standardized by means of the RFC3501

POP3 heir
- Data storing and email processing on the remote server
- Email are not loaded locally by the client

Request-Reply model
- The client starts each activity sending a request
- The server generates a reply carrying data required by client's query

IMAP4 Session
- FSM client sends commands in some specific order
- When request are successful, client enters in a new state
Architecture of an IMAP Client Library

• **Basic IMAP client library architecture:**
  - **Communication Layer**, handles socket connectivity.
  - **Low-level IMAP protocol handler**, managing request-reply communication (tags handling and untagged/tagged response identification)
  - **IMAP interpreter**, parsing role (transforms server’s reply into programming language native types)
  - **IMAP FSM**, handles IMAP’s FSM (manages state transitions)
  - **IMAP high-level interface**, implements the various IMAP commands by means of IMAP low-level interface’s services and IMAP parsing.
Evaluation Criteria (1/2)

Number of source lines of code (SLOC)
- No blank lines and comments
- Count the numbers of lines of all library's source files
- “The more the SLOC the more the time required to write the software and to test it.”

Functionality of primitives
- Qualitative parameters...
- …to make sense of the capability of the library to provide a complete and transparent implementation of the IMAP commands
- Related to the number and type of function/method calls

Evaluation Criteria (2/2)

Memory
- Performance parameter
- Expresses the memory usage in a language specific IMAP library at run-time
- Test program: logging and fetching a bunch of messages

Execution-Time
- Performance parameter
- Test program: executing different commands and measuring time required by each library to perform its activities.
Three modules:
- im_client (front-end interface)
- scanning (lexical analysis)
- parsing (parses server's response)

- im_client module's outcome: {ResultValue ('ok', 'not' or 'bad'), ParsedReply}, ParsedReply is the server's reply data
- Check socket connection and re-instantiate it if needed, preventing brutal closing
Evaluated Solutions (3/5)

C# Library

- Uses native library of .Net platform
- Three main source files:
  - ImapBase.cs (handles low-level client-server connection),
  - Imap.cs (protocol's engine, handles IMAP commands),
  - ImapException.cs (manages protocol's fault and internal error)
- Custom policy to fetch email message, three methods:
  - FetchPartHeader fetches only the Header section of the original email message,
  - FetchPartBody fetches only the Body section of the original email message,
  - FetchMessage performs a complete parsing and retrieve the whole message, generates an XML file such as output of its parsing activity. Client program must re-interpreter XML file to get meaningful data.

Evaluated Solutions (4/5)

Python Library

- Transparency in function declaration: 1:1 mapping of IMAP commands to methods.
- Authentication mechanism supported (IMAP AUTHENTICATE command)
- Client program must create the IMAP command (library manages Tag numbering only)
- No Parsing...client program must parse server's reply
- All methods returns a tuple: \{ Tagged response, UnTagged Response \}
Evaluated Solutions (5/5)

Ruby Library

- Transparency in function declaration: 1:1 mapping of IMAP commands to methods.
- Authentication mechanism supported (IMAP AUTHENTICATE command)
- Client program must create the IMAP command (library manages Tag numbering only)
- Partial Parsing activity, only FETCH
  /BODYSTRUCTURE/BODY[TEXT]/BODY[section]/ENVELOPE replies are parsed
- All methods returns a native Ruby types containing query's sequence_number and a tuple
  [key, value], the former is FETCH command keyword(FLAGS...TEXT) whereas the latter
  is associated data.

Results - SLOC (1/4)

<table>
<thead>
<tr>
<th>Languages</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erlang</td>
<td>1,189</td>
</tr>
<tr>
<td>Python</td>
<td>472</td>
</tr>
<tr>
<td>Ruby</td>
<td>1,612</td>
</tr>
<tr>
<td>C#</td>
<td>1,089</td>
</tr>
<tr>
<td>Java</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Erlang and Ruby perform a full parsing of server's replies:

- Erlang Parser counts 818 lines of code
- Ruby Parser counts 1048 lines of code
### Results - Functionality of primitives (2/4)

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

### Results - Amount of memory (3/4)

<table>
<thead>
<tr>
<th>Language</th>
<th>Total</th>
<th>Code</th>
<th>Data/Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erlang</td>
<td>8,056</td>
<td>2,868</td>
<td>4,656</td>
</tr>
<tr>
<td>Python</td>
<td>6,748</td>
<td>4,400</td>
<td>1,684</td>
</tr>
<tr>
<td>Ruby</td>
<td>14,942</td>
<td>4,332</td>
<td>10,386</td>
</tr>
<tr>
<td>C#</td>
<td>18,800</td>
<td>11,148</td>
<td>2,748</td>
</tr>
<tr>
<td>Java</td>
<td>212,852</td>
<td>14,428</td>
<td>190,060</td>
</tr>
</tbody>
</table>
**Results - Execution Time (4/4)**

<table>
<thead>
<tr>
<th>Command/Primitive</th>
<th>Erlang</th>
<th>Python</th>
<th>Ruby</th>
<th>C#</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGIN</td>
<td>28.6</td>
<td>45.7*</td>
<td>26.1</td>
<td>32.7</td>
<td>30.7</td>
</tr>
<tr>
<td>SELECT</td>
<td>2.5</td>
<td>2.1*</td>
<td>44.3</td>
<td>13.2</td>
<td>4.5</td>
</tr>
<tr>
<td>FETCH BODY[TEXT]</td>
<td>43.0</td>
<td>41.1*</td>
<td>80.2</td>
<td>40.0</td>
<td>40.2</td>
</tr>
<tr>
<td>FETCH BODY[HEADER]</td>
<td>2.5</td>
<td>0.2*</td>
<td>44.5</td>
<td>5.3</td>
<td>2.9</td>
</tr>
<tr>
<td>FETCH BODY[HEADER:FIELDS]</td>
<td>2.3</td>
<td>0.2*</td>
<td>43.0</td>
<td>n/a</td>
<td>1.3*</td>
</tr>
<tr>
<td>FETCH BODYSTRUCTURE</td>
<td>1.8</td>
<td>0.2*</td>
<td>80.2</td>
<td>n/a</td>
<td>1.2*</td>
</tr>
<tr>
<td>FETCH (ENVELOPE SIZE: BODY:PEEK:HEADER:READY)</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>1.9</td>
</tr>
<tr>
<td>LIST</td>
<td>0.7</td>
<td>0.9*</td>
<td>52.1</td>
<td>n/a</td>
<td>6.3</td>
</tr>
</tbody>
</table>

* = command executed without any parsing of the response

---

**Results Summary**

**Full Parsing (Erlang Library vs Ruby Library)**
- Erlang implementation outdoes Ruby library features, indeed its performances (such as Memory consumption, Execution-time) are better than values estimated for Ruby.
- Erlang library counts less lines of code of Ruby implementation, as demonstrated through SLOC analysis.

**Partial Parsing (Java Library vs C# Library)**
- Java implementation consumes huge amount of memory.
- Execution-time are comparable even if Java has better performances in the FETCH command handling, probably it depends on internal proprietary implementation.

**NO Parsing (Python Library)**
- Python has best performances (memory consumption and execution-time) but they depend on the lack of parsing activity and Python JVM characteristics.
Conclusions

The aim is...
- ...to evaluate Erlang ability to meet requirements of productivity and performance for a distributed system

We conclude that the Erlang library...
- ...can deliver the requirements of functionality and performance for a distributed system
- has high execution-time without high memory cost