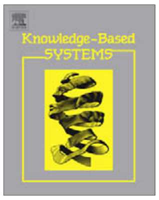
Knowledge-Based Systems 22 (2009) 516–522

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys



A user-extensible and adaptable parser architecture ☆

John Tobin *, Carl Vogel

School of Computer Science and Statistics, Trinity College, Dublin 2, Ireland

ARTICLE INFO

Article history:
Available online 19 January 2009

Keywords:
Parsing
Postfix
Anti-spam
Log files

ABSTRACT

Some parsers need to be very precise and strict when parsing, yet must allow users to easily adapt or extend the parser to parse new inputs, without requiring that the user have an in-depth knowledge and understanding of the parser's internal workings. This paper presents a novel parsing architecture, designed for parsing Postfix log files, that aims to make the process of parsing new inputs as simple as possible, enabling users to trivially add new rules (to parse variants of existing inputs) and relatively easily add new actions (to process a previously unknown category of input). The architecture scales linearly or better as the number of rules and size of input increases, making it suitable for parsing large corpora or months of accumulated data.

1. Introduction

The architecture described herein was developed as part of a larger project to improve anti-spam defences by analysing the performance of the set of filters currently in use, optimising the order and membership of the set based on that analysis, and developing supplemental filters where deficiencies are identified. Most anti-spam techniques are content-based (e.g. [6,3,10]) and require the mail to be accepted before determining if it is spam, but rejecting mail during the delivery attempt is preferable: senders of non-spam mail that is mistakenly rejected will receive an immediate non-delivery notice; resource usage is reduced on the accepting mail server (allowing more intensive content-based techniques to be used on the mail that is accepted); users have less spam mail to wade through. Improving the performance of anti-spam techniques that are applied when mail is being transferred via Simple Mail Transfer Protocol (SMTP)[1] is the goal of this project, by providing a platform for reasoning about anti-spam filters. The approach chosen to measure performance is to analyse the log files produced by the SMTP server in use, Postfix [12], rather than modifying it to generate statistics: this approach improves the chances of other sites testing and using the software. The need arose for a parser capable of dealing with the great number and variety of log lines produced by Postfix: the parser must be designed so that adding support for parsing new inputs is a simple task, because the log lines to be parsed will change over time. The variety in log lines occurs for several reasons:

- Log lines differ amongst versions of Postfix.
- The mail administrator can define custom rejection messages.
- External resources Postfix is configured to use (e.g. DNS Black List or policy servers [13]) can change their messages without warning.

It was hoped to reuse an existing parser rather than writing one from scratch, but the existing parsers considered were rejected for one or more of the following reasons: they parsed too small a fraction of the log files; their parsing was too inexact; they did not extract sufficient data. The effort required to adapt and improve an existing parser was judged to be greater than the effort required to write a new one, because the techniques used by the existing parsers severely limited their potential: some ignored the majority of log lines, parsing specific log lines accurately, but without any provision for parsing new or similar log lines; others sloppily parsed the majority of log lines, but were incapable of distinguishing between log lines of the same category, e.g. rejecting a mail delivery attempt. The only prior published work on the subject of parsing Postfix log files that the authors are aware of is *Log Mail Analyser: Architecture and Practical Utilizations* [4], which aims to extract data from log files, correlate it, and present it in a form suitable for a systems administrator to search using the myriad of standard Unix text processing utilities already available. A full state of the art review is outside the scope of this paper but will be included in the thesis resulting from this work.

The solution developed is conceptually simple: provide a few generic functions (*actions*), each capable of dealing with an entire

☆ Supported by Science Foundation Ireland RFP 05/RF/CMS002.

* Corresponding author.
E-mail addresses: tobinjt@cs.tcd.ie (J. Tobin), vogel@cs.tcd.ie (C. Vogel).

[1] Simple Mail Transfer Protocol transfers mail across the Internet from the sender to one or more recipients. It is a simple, human readable, plain text protocol, making it quite easy to test and debug problems with it. The original protocol definition is RFC 821 [9], updated in RFC 2821 [8].

Table 1
Similarities with ATN.

ATN	Parser architecture	Similarity
Networks	Framework	Determines the sequence of transitions or actions that constitutes a valid input
Transitions	Actions	Assembles data and imposes conditions the input must meet to be accepted as valid
Abbreviations	Rules	Responsible for classifying input

category of inputs (e.g. rejecting a mail delivery attempt), accompanied by a multitude of precise patterns (*rules*), each of which matches all inputs of a specific type and only that type (e.g. rejection by a specific DNS Black List). It is an accepted standard to separate the parsing procedure from the declarative grammar it operates with; part of the novelty here is in the way that the grammar is itself partially procedural (each action is a separate procedure). This architecture is ideally suited to parsing inputs where the input is not fully understood or does not conform to a fixed grammar: the architecture warns about unparsed inputs and other errors, but continues parsing as best it can, allowing the developer of a new parser to decide which deficiencies are most important and require attention first, rather than being forced to fix the first error that arises.

2. Architecture

The architecture is split into three sections: framework, actions and rules. Each will be discussed separately, but first an overview:

Framework: The framework is the structure that actions and rules plug into. It provides the parsing loop, shared data storage, loading and validation of rules, storage of results, and other support functions.
Actions: Each action performs the work required to deal with a single category of inputs, e.g. processing data from rejections.
Rules: The rules are responsible for classifying inputs, specifying the action to invoke and the regex that matches the inputs and extracts data.

For each input the framework tries each rule in turn until it finds a rule that matches the input, then invokes the action specified by that rule.

Decoupling the parsing rules from their associated actions allows new rules to be written and tested without requiring modifications to the parser source code, significantly lowering the barrier to entry for casual users who need to parse new inputs, e.g. part-time systems administrators attempting to combat and reduce spam; it also allows companies to develop user-extensible parsers without divulging their source code. Decoupling the actions from the framework simplifies both framework and actions: the framework provides services to the actions, but does not need to perform any tasks specific to the input being parsed; actions benefit from having services provided by the framework, freeing them to concentrate on the task of accurately and correctly processing the information provided by rules.

Decoupling also creates a clear separation of functionality: rules handle low level details of identifying inputs and extracting data; actions handle the higher level tasks of assembling the required data, dealing with the intricacies of the input being parsed, complications arising, etc.; the framework provides services to actions and manages the parsing process.

Some similarity exists between this architecture and William Wood's Augmented Transition Networks (ATN) [15], used in Computational Linguistics for creating grammars to parse or generate sentences. The resemblance between the two (shown in Table 1. (Similarities with ATN)) is accidental, but it is obvious that the two approaches share a similar division of responsibilities, despite having different semantics.

2.1. Framework

The framework takes care of miscellaneous support functions and low level details of parsing, freeing the programmers writing actions to concentrate on writing productive code. It links actions and rules, allowing either to be improved independently of the other. It provides shared storage to pass data between actions, loads and validates rules, manages parsing, invokes actions, tracks how often each rule matches to optimise rule ordering (Section 3.2), and stores results in the database. Most parsers will require the same basic functionality from the framework, plus some specialised support functions. The framework is the core of the architecture and is deliberately quite simple: the rules deal with the variation in inputs, and the actions deal with the intricacies and complications encountered when parsing.

The function that finds the rule matching the input and invokes the requested action can be expressed in pseudo-code as:

```
for each input:
  for each rule defined by the user:
    if this rule matches the input:
      perform the action specified by the rule
      skip the remaining rules
      process the next input
  warn the user that the input was not parsed
```

2.2. Actions

Each action is a separate procedure written to deal with a particular category of input, e.g. rejections. The actions are parser-specific: each parser author will need to write the required actions from scratch unless extending an existing parser. It is anticipated that parsers based on this architecture will have a high ratio of rules to actions, with the aim of having simpler rules and clearer distinctions between the inputs parsed by different rules. In the Postfix log parser developed for this project there are 18 actions and 169 rules, with an uneven distribution of rules to actions as shown in Fig. 1. Unsurprisingly, the action with the most associated rules is `DELIVERY_REJECTED`, the action that handles Postfix rejecting a mail delivery attempt; it is followed by `SAVE_DATA`, the action responsible for handling informative log lines, supplementing the data gathered from other log lines. The third most common action is, perhaps surprisingly, `UNINTERESTING`: this action does nothing when executed, allowing uninteresting log lines to be parsed without causing any effects (it does not imply that the input is ungrammatical or unparsed). Generally rules specifying the `UNINTERESTING` action parse log lines that are not associated with a specific mail, e.g. notices about configuration files changing. The remaining actions have only one or two associated rules: some actions are required to address a deficiency in the log files, or a complication that arises during parsing; other actions will only ever have one log line variant, e.g. all log lines showing that a remote client has connected are matched by a single rule and handled by the `CONNECT` action.

Using the `CONNECT` action as an example: it creates a new data structure in memory for the new client connection, saving the data extracted by the rule into it; this data will be entered into the database when the mail delivery attempt is complete. If a data struc-