FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Petr Gebauer

Log analyser

Department of Applied Mathematics

Supervisor of the bachelor thesis:	Martin Mareš
Study programme:	Computer Science
Study branch:	General Conputer Science

Prague 2021

I would like to thank Mgr. Martin Mareš Ph.D. for his guidance, a ton of helpful advice, and the patience he displayed during our cooperation on this thesis.

Title: Log analyser

Author: Petr Gebauer

Department: Department of Applied Mathematics

Supervisor: Martin Mareš, Department of Applied Mathematics

Abstract: This thesis sets out to create a tool for analyzing logs based on a configuration supplied by the user. Some examples of possible a analysis include restructuring (parts of) the logs so they are easier to read for humans or machines, detecting events in the logs or calculating stats from them. The main goals for the tool are its flexibility and sufficient performance for processing of real–life logs in reasonable time.

Keywords: logs analysis parsing

Contents

Introduction

Computer systems write a large amount of information about their activity into what people call logs. Logs often provide the primary means of assessing a system's state. However, their analysis poses a number of problems since they are usually neither easy to read directly nor simple to process automatically.

One huge challenge for humans reading logs is the sheer size of the logs. This problem affects automatic processing, too. As computing power grows with passing time, so does the power generating the logs. Logs may also be generated by multiple instances, each with its own hardware and forwarded for processing to a single machine. Also, since log processing is usually only a tool for performing analysis of the system's main activity, it can only be allocated limited resources.

This thesis aims to create a tool for reading logs and extracting the important information from them in a format that can be either read by humans or easily used for further automated processing. This tool will need to be highly flexible to process logs with different structures and fast enough to stand up to the challenge posed by the size of production logs.

1. Logs and their structure

Logs are usually either written directly to a file by the process that generates them or passed to a dedicated program. This program may do some simple alteration of the log messages it is given and either store them in one or multiple files on the same machine or send them over the network. There is a protocol for this network communication called syslog.

We want to read logs both from files and from a socket using the syslog protocol. Furthermore, the format of log files usually stems from the structure specified by syslog standards.

1.1 Syslog protocol and structure

Syslog is a protocol for handling log messages and sending them over the network. It is also used by syslog daemons for logging messages on the same machine where they originated. The protocol forces some message structure, so even messages stored in a log file have a similar format. Therefore, it is useful to familiarize ourselves with parts of the syslog RFCs.

The original syslog standard was RFC 3164 [14], which was later obsoleted by RFC 5424 [11]. The architecture specified by these RFCs allows messages to be generated (machines capable of that are called *devices* or *originators* based on the particular RFC), forwarded (by so-called *relays*) and collected (by *collectors*, also known as *syslog servers*). A device or a relay may send messages to one or more relays and collectors (without knowing whether a particular machine is a relay or a collector). A relay will send all or some of the messages it receives further. It may also generate and send its own messages, in which case it is also acting as a device.

1.1.1 RFC 3164

RFC 3164 [14] has been obsoleted by RFC 5424 [11] but is still in common use. It specifies the following format :

"<" *prival* ">"	decimal number consisting of three digits containing facility and severity values. The standard specifies that facility values must be in the range from 0 to 23 inclusive and severity from 0 to 7 inclusive. It also specifies the mapping of values to their meaning (see below)
timestamp	timestamp in the format "Mmm dd hh:mm:ss" where Mmm are three characters denoting the month of the year (e.g., `Jan`, `Feb`, `Dec`) and other parts are day of month, hour, minute and second. They are represented by two digits (in case the number is at least 10) or a single space and one digit.
sp	space
hostname	hostname or IP address of the machine

sp	
msg	the actual text message

The prescribed meanings of facility values are:

0	kernel messages
2	mail messages
3	system daemons
4	security/authorization messages
11	FTP daemon

While the specified severity value meanings are:

0	Emergency: the system is unusable
1	Alert: action must be taken immediately
2	Critical
3	Error
4	Warning
5	Notice: a normal but significant condition
6	Informational
7	Debug

The priority value is then calculated as $8 \cdot \mathit{facility} + \mathit{severity}$.

This format is required for messages sent by relays and recommended for messages sent by the original devices. If a relay receives a message that does not comply with this format, it must modify such a message before sending it further.

An example of a valid timestamp according to this RFC would be `Oct 11 22:14:15`. An example of a whole message would be

```
<34>Oct 11 22:14:15 mymachine su: 'su root' failed for lonvick
```

1.1.2 RFC 5424

RFC 5424 [11] is the newer of the two and it is more strict about message structure, so logs conforming to it are easier to parse. On the other hand, it is perhaps less frequently used. This standard specifies that messages logged using the syslog protocol must have the following format:

“<” *prival* “>”	priority value as described in the section about RFC 3164 except that the facility and severity values are not mandatory, only mentioned as common.
version	version of the syslog protocol
sp	space
timestamp	timestamp with format derived from RFC 3339 [15]

	with further restrictions (see below)
sp	
hostname	
sp	
app-name	
procid	usually process id of the syslog system (see the RFC for more details)
msgid	identifier of the type of the message, for example `TCPIN` for messages about incoming TCP traffic
structured-data	
[*sp msg*]	optional space-separated text message

Some of the elements (for example *structured-data*) may be replaced by "`-`".

Timestamps are required to conform to RFC 3339 [15], which stems from ISO 8601 [12]. The format is as follows:

date-fullyear	year expressed using all four digits, e.g., 1970
`-`	
date-month	two digits representing the number of the month in the year starting from 01
`-`	
date-mday	day of the month written as two digits starting from 01
`T`	
time-hour	
`:`	
time-minute	
`:`	
time-second	
["." time-secfrac]	optional dot-separated fraction of the second
time-offset	either "Z" for UTC or +/-hh:mm representing local time offset (calculated as local time minus UTC), UTC can also be expressed as "+00:00" (or "-00:00", which has a special meaning)

There are some additional restrictions on the timestamp format. RFC 3339 allows the "T" and "Z" characters to be lowercase (i.e., "t" and "z"), "T" may be omitted, but RFC 5424 prohibits this. In addition, RFC 5424 specifies that leap seconds must not be used.

An example of a correct timestamp is `2003-08-24T05:14:15.000003-07:00` representing 24th of August 20003 5:14:15 AM and 3 microseconds in a timezone 7 hours behind UTC (so it is 12:14:15 UTC).

The *structured-data* element contains either "`-`" or one or more "structured data elements" (*sd-element*-s). Each *sd-element* contains a unique identifier of the kind of the element and one or more name, value pairs.

The MSG element is a text encoded using UTF-8 following RFC 3629 [19] but with no prescribed structure. It must start with byte order mark (denoted BOM in the example below).

The following is an example of a valid message:

```
<165>1 2003-10-11T22:14:15.003Z mymachine.example.com evntslog - ID47
[exampleSDID@32473 iut="3" eventSource="Application"
eventID="1011"] BOM An application event log entry
```

Note in particular the addition of the structured data element and the format of the timestamp is very different from RFC 3164.

Message order

Log messages from different sources (be it different machines or running processes) can and often do intertwine, which complicates reading. Furthermore, even messages from a single machine and a single source may contain information about many simultaneous connections. Parallel activity on these connections results in log messages getting mixed, and as a result, it is difficult for a person to trace activity on a single connection.

Different format

Another problem for automatic processing is the relatively free format of the messages. This problem is especially significant with RFC 3164, which has no *structured-data* element. However, even with RFC 5424 and if the *structured-data* element is used, the *msg* part of the message can have any internal structure as it can often be generated by a private application.

Timestamps

In addition to having different formats, timestamps without a specification of their offset from UTC can be ambiguous. Even if all of the messages are from the same timezone, when daylight saving time ends, there are timestamps that could represent two different times based on whether they are using daylight saving time or not (and this fact cannot be derived from the timestamps themselves).

1.2 Log storage

Logs are typically stored in multiple files, which are created as time passes. A new file is created either after a fixed time has elapsed or when the previous file exceeds certain size. What is critical for our processing, these files' names are usually either dependent on the time of creation and remain constant after the file was created, or the files are *rotated*.

With rotation, the files have numbers at the end of their name. The one to which (most of) the writing takes place will usually have no number, which is interpreted as 0. The next older file will have number 1, and others will have numbers 2, 3, etc. When a new file is created, all of the existing files get their numbers increased by one (the oldest ones possibly getting deleted) and the new file is created with no numeric suffix. All processes logging into this set of files then receive SIGHUP signal and reopen their log file (thus starting to log to the new one).

As a result, some of the processes may still be logging to the old file, while others have already reopened their file and are logging into the new one. Moreover, because of the changes in numeric filename suffixes, a file name is not a reliable identifier of the file itself. It is also common for some of the oldest files (often all but the two newest) to get compressed.

2. Solution principles

2.1 Goals

Let us discuss a few examples of the kind of log we might want to process.

- ssh logs, which contain information about users accessing our system remotely through the ssh protocol. Their lines may look like this:

  ```
  Apr 30 08:42:07 myhost sshd[5933]: Accepted publickey for pg from
  192.168.1.1 port 45740 ssh2:
  RSA SHA256:EZwkd8q58UapKekDC48ZcNvfSVy1NMqZagZqtCUZeHymf1BaUAFT6
  ```

  The ssh daemon may very well be set up to log to authentication logs, in which case the messages from it would be mixed with other messages, such as:

  ```
  Apr 30 08:44:05 myhost sudo:    pg : TTY=pts/2 ; PWD=/home/pg
  ; USER=root ; COMMAND=/usr/bin/less /var/log/syslog
  ```

- Postfix logs. Postfix is an implementation of a mail server. Its looks may look like so:

  ```
  2020-04-04T06:59:43.000422+02:00 Postfix/smtp[8573]:
  7BE3C1C2F52: to=<pca@wwilde.isageek.de>, relay=none,
  delay=145481, delays=145450/0.06/30/0, dsn=4.4.1, status=deferred
  (connect to wwilde.isageek.de[213.95.21.121]:25: Connection timed
  out)
  ```

- Web server logs. The following is a line from the error log of a web server called apache2:

  ```
  [Fri Apr 30 07:37:34.329024 2021] [mpm_event:notice]
  [pid 1491:tid 140579503806336] AH00489: Apache/2.4.18 (Ubuntu)
  configured -- resuming normal operations
  ```

  And this one is from the access log of the same server:

  ```
  192.168.1.41 - - [30/Apr/2021:09:23:15 +0200] "GET /ABCD/
  HTTP/1.1" 200 1175 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64;
  rv:88.0) Gecko/20100101 Firefox/88.0"
  ```

- Logs of some programs we developed for our internal usage. These logs may have pretty much any structure.

We can see that the structure of the logs differs significantly. They tend to have a timestamp, some sort of a specification of who is responsible for the message (like hostname and process id) and the message itself, but that is where the similarities end. This means that our program will need to be very flexible and let its user specify the structure of their log rather than trying to figure it out itself.

Let's now focus on the Postfix logs in our discussion and see a few examples of how we might want to process them.

- As an elementary example, we could detect messages talking directly about an e-mail (referring to it by an ID like `7BE3C1C2F52`). These messages tend to get lost in many other messages and, also messages talking about different e-mails will get intertwined among themselves. Therefore we would like to filter such messages and change their order. We would like to print messages associated with one particular e-mail chronologically, followed by those regarding another e-mail and so on.

- For each minute, count the number of e-mails sent or received in that minute.

- Count the number of "connection lost" messages for each server we comunicate with and print those where the number exceeds a certain threshold.

- Keep track of how many e-mails are received from each third-party server and alert about very high values.

- Watch how e-mails are forwarded to detect when a user has set up a forwarding loop.

These requirements are again rather diverse. Even in cases where we want to output a number of some occurrences, it might not be as simple as incrementing a counter. This is because a line may indicate multiple occurrences. Therefore we also need to be flexible in how we process the logs once we understand their structure. Let us pick grouping by mail ID as the running example.

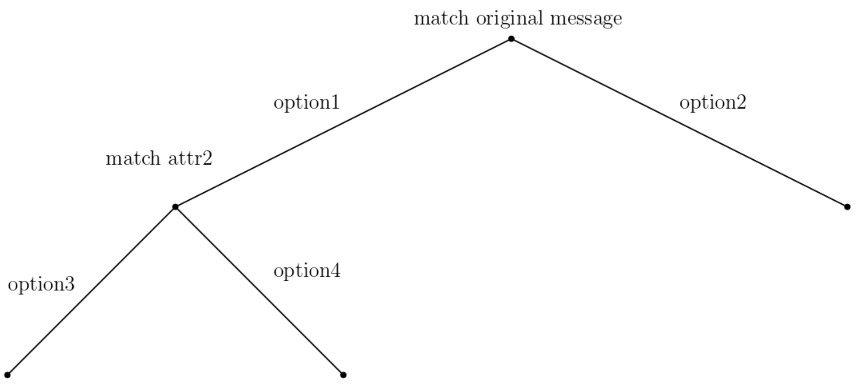
If we have implemented the grouping, it should be possible to implement the other examples with little effort. Namely, it should be possible to describe the syntax of the messages just once and re-use this description later.

More generally, the processing should be split into a parsing part, which will be specific for the log, and an analysis, which will specify what should be done with the log. Then, when we wish to add a feature to our log analysis, we only need to update the analysis. Also, if the log structure changes somewhat, we will only need to change the syntax analysis.

Furthermore, it should be possible to write a more general part of the parsing first and then focus on the details. This will allow us to have some parts of the configuration shared among multiple similar use cases. For example, we may have all our machines configured to use the same timestamp format and to follow RFC 3164, and we might later want to extend our configuration to process logs with messages from services other than Postfix mixed in. In that case, it would be helpful first to parse the timestamp, detect the service responsible for the message and isolate the free-formed text message (the *msg* part as specified by the syslog protocol RFCs). Then based on the source of the message, we would run it through the appropriate parsing machinery for that particular service.

To allow the next stage of the parsing to take place without having to redo the parsing done by previous stages, there needs to be a way of accessing the results of previous stages. Since in many situations, there is no single clear output from parsing (even from a single stage), more than one such "outputs" need to be accessible at once. We call them *attributes*.

The former stages should also have the capability to decide what parsing schemes are used going forward. For example, when the first stage detects the source of the message, the next stage needs to be chosen based on what the source was. The flow control then takes the form of a tree where the root does top-level parsing and decides which of its child nodes should be invoked next. As one of its child nodes gets invoked, it also decides which of the nodes under it should be used next and so on down to the leaves. The tree could look something like this:



It would be useful to be able to have a common structure for parsing a part of a message and use it in multiple configurations. For example, a timestamp is likely to look the same in many different logs, so its parsing tree could be written and debugged only once and then re-used. Also, if a single node's specification is not trivial, we should support reusing parts of nodes if possible.

In the following few sections, we will give a high-level overview of our implementation of the parsing before discussing analysis.

2.2 Basic parsing

There needs to be a way for the user to specify how to parse attributes of a particular and when to descend to which subtree below it. The chosen configuration language should be flexible enough to allow for parsing very different kinds of input but simple enough to allow practical usage. Regular expressions have been used for this kind of specification since they are strong enough for many use cases while still being intuitive and easy to use. For example, if we wanted an expression matching timestamps conforming to the requirements of RFC 5424, we could write:

```
[0-9]{4}-[01][0-9]-[0-3][0-9]T[0-2][0-9]:[0-5][0-9]:[0-6][0-9]
(\.[0-9]+)?
```

There is also the question of how to assign to attributes and reference them. The mechanism should be able to assign to multiple attributes at once (or fail to match entirely), and it should be able to adapt to differences in the matched text. The chosen approach is somewhat similar to back-references used by a standard text processing tool named sed. In the case of sed, the user can mark parts of the regular expression and reference them later by a backslash and a number referring to its position.

However specifying attributes by a number would significantly reduce readability, especially since the number would have to identify both the node and the part of the expression in that node. This approach would also defeat the purpose of library trees since they could not use attributes set by the tree calling it without knowing the caller's structure.

The other available solution is to reference attributes by some fixed key: we chose textual names.

We did not want to write the matching ourselves, so we needed some library to do the work for us. We needed it to be easy to connect with the rest of our program (which is written in C++), and we also wanted it to support the textual names discussed earlier. These are some possible candidates for such a library:

- POSIX regular expressions — These are the regular expressions offered by the standard C library. They don't seem to support named captures (at least we found no support for it).

- `std::regex` from the standard C++ library [2] — This library also seems to lack support for named captures.

- Python regular expression library [17] — It does support named captures but isn't easy to integrate with the rest of our program. Although Python has a C API, to use this library, we would probably have to create a Python string every time we need to do the matching. This would mean allocating memory for that string and copying its contents from our buffer to the allocated block. That would likely severely slow down the execution.

- PCRE [16] — This library supports named captures, and it is easy to integrate with the rest of the program.

- boost regular expressions [13] — It supports named captures and is intended for usage from C++.

- Grok [18] — This is actually a tool for parsing logs, which offers a library with C interface performing the matching we need. It uses PCRE as its back end, supports named captures, and offers other features. One such feature is support for storing named patterns in a separate file and using them by referring to their name. This feature was considered good for configuration re-usability. There are also other features, such as encoding a piece of text as a JSON string.

We chose grok because of its additional features.

Let's look at an example of using regular expressions with named captures for attribute assignment. We could take the expression above and modify it to detect and separate the timestamp in each message. In grok's syntax, we can write:

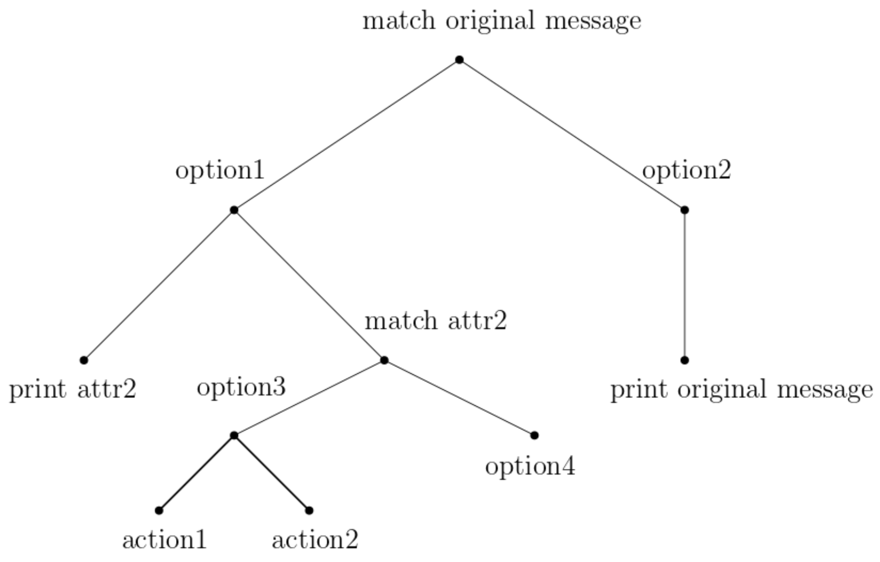
```
^%{time=[0-9]{4}-[01][0-9]-[0-3][0-9]T[0-2][0-9]:[0-6][0-9]:[0-6][0-9]
(\.[0-9]+)?} %{message=.*}
```

The timestamp is then stored in the attribute `time`, while the rest of the log message is stored in `message`.

To support more complex parsing(for example for interfacing with analysis on different levels), and in part for deciding what parsing should be done, the tree contains two kinds of nodes — actions and options. One particular action is a match, which specifies (by name) a single attribute to be matched and has options (the other kind of nodes) as its children. An option specifies a regular expression, and it can have actions (so in particular matches) as its children.

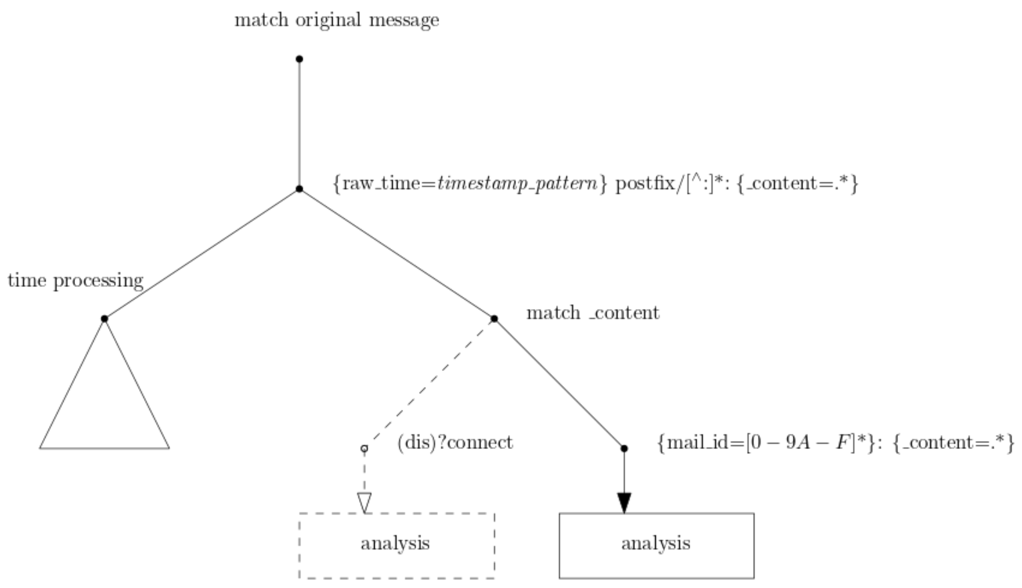
When our program arrives at a match node, call it M, it tries to match the specified attribute against all the regular expressions in its children one by one. Every time this match is successful, the program descends into the subtree under that option before continuing to other options under M. When the program descends into a subtree under an option, all of its child action are performed one by one. The tree is, therefore, walked through in a depth-first "left-to-right" manner, where the children of each node are ordered in the same way they appeared in the configuration.

The tree with two kinds of nodes looks like this (the prints are another kind of actions):



In the beginning, there is only one attribute filled and it contains the original message. As the attributes are added, they can all be used for matching. So, for example, a match at a higher level may use attributes defined at a lower level. These attributes could have been assigned to already since the tree is walked through in a depth-first manner. This behavior is also beneficial for library trees described below. Attributes that haven't been assigned any value will appear to be empty.

In our example, the tree may look like the figure on the next page. If we later also wish to keep track of connects and disconnects for example, we may add the dashed branch in the image and modify analysis to reflect the addition.


match original message
{raw_time=*timestamp_pattern*} postfix/[^:]*: {_content=.*}
time processing
match _content
(dis)?connect
{mail_id=[0 − 9A − F]*}: {_content=.*}
analysis
analysis

2.3 Re-usability

One repeating part of the parsing specification is the regular expression patterns. For example, a pattern matching an IP address could be used in multiple kinds of logs, so these are ideal candidates for parts of parsing nodes to be re-used. We allow for patterns to be named, stored separately from the parsing tree structures, and referred to by their name.

In order to re-use entire trees, the user can define multiple trees. One of them will be marked main and parsing will start there. Other trees can be used as libraries. To use a different tree, the user can create a dedicated action (named parse_by), which will call that tree. The tree then has access to all of the attributes.

2.4 Semantic processing

The primary consideration with analysis was its flexibility. It is rather hard to predict what operations the user will want to perform and design and optimize the mechanism for it. For example, we may choose to support aggregation by a key stored in an attribute. This way, the user could parse mail log messages in such a way to get e-mail identifiers in one attribute named `mail_id` and wish to count the total number of e-mails mentioned in the log. He would specify that messages should be aggregated by `mail_id` and counted. In another use case, some other attribute could be treated as a numeric one and the numbers would be added up.

Another option could be to group the e-mails by this key (`mail_id`), possibly sorted by another attribute and printed. However, this still supports only a limited number of operations we could think of and it requires the key to be precisely the same. It does not allow for grouping (or other kinds of aggregation) of messages whose keys aren't exactly the same. Furthermore, we might like to support conditions being formulas with logical operators (of arbitrary complexity), conditions containing numerical comparison, different kinds of sorting which consider multiple attributes and other features. So implementing all we can think of would be very demanding, and even that would likely be too rigid for many use cases. The user may also want to recognize something more complex than a regular language, or to assign values that are not present in the input to attributes.

The decision was therefore taken to integrate into the project an existing programming language that could be used through a suitable interface at the time of parsing a message. The control could therefore go back and forth between parsing and the user-supplied program, which would, besides complex analysis, allow for some features to be generated using the user program and then influence the behavior of the parsing part.

There are two obvious ways to supply the user's code — either in designated files which may also be viable stand-alone programs, or embedded in the configuration. The former approach has the advantages of separating parsing and analysis more and allowing (at least in some cases) some degree of debugging and testing the code separately without the need to run it back-to-back integrated with the project. Also, large complex programs would overshadow the rest of the configuration, making it hard to read. However, some operations may only require a single short line of code, in which case it may be better to write them directly into the config file. Therefore support for both was implemented.

The user may write their own functions and call them from the parsing tree. A new action node was introduced for this, named *call* — the decision to call a function is therefore dependent on whether a certain option matches a certain attribute. The function called receives all of the attributes in a structure called *context*, where the attributes are simply referred to by name. The function can then modify the context by changing values of existing attributes or adding completely new attributes, execute any code (which can, for example, communicate over the network, write to files, etc.), and call trees (just like the parse_by action).

Inline user code can also be written directly into the configuration file through a dedicated action: *call*. It can access and modify the context just like functions can. This code is executed again every time and therefore can hardly store any information about one log message and use it when parsing a different message. In addition, it may need to interact with more complex logic written in a separate file. To address these issues, every piece of inline code gets assigned a user-supplied source file and it is executed as if it were a part of that file. It can call functions from this file, access its global variables, etc.

When choosing the language for user programming, user comfort was given a high priority. We also needed the language to be easy to interface with the core of the program, which is written in C++. Python and Lua were the languages on the shortlist, and Python was finally chosen because it is very well known and famous for its wealth of libraries. Also, in the beginning, there was some thought given to using machine learning for processing and diagnosing logs, for which Python is very useful. Although it quickly became clear this functionality would be out of the scope of this thesis, it is still regarded as a possible future extension.

2.4.1 Sessions and time parsing

We expected many use cases to require keeping track of sessions of some kind. An example could be processing of ssh logs and monitoring how often and when each user logs in and for how long. Or parsing Postfix logs to identify log lines associated with a single e-mail (in the simplest case printing them grouped together like in the example at the beginning of the chapter).

To be able to deal with incomplete logs, we should be prepared for the possibility that some sessions may have their initialization but not their termination mentioned in the log. For this reason, we support session timeout.

The session support we implemented offers storing user-defined objects with a textual identifier, and these objects outlive the function call in which they were registered. The session may be an instance of any class derived from a built-in class for sessions. To allow for handling different kinds of sessions in the same configuration file and the same module without identifier collision, sessions are scoped by their class.

When registering a session, a reference time and a timeout can be specified, in which case the session will be terminated at *reference_time+timeout* unless the user indicates there has been activity on it. If any activity is reported, the reference time is updated - the session, therefore, times out after *timeout*-long period of inactivity.

Since we timeout sessions ourselves, instead of leaving it up to the user, we need to parse timestamps from the log into a representation where the elapsed time between two timestamps can be calculated. However, depending on the particular log, timestamps may have different formats and be located in different parts of the message. The user, therefore, needs to be able to specify the timestamp attribute and format. As discussed in the first chapter, timezone support is also important — not only for processing logs with messages from different zones but also for disambiguating timestamps around the end of daylight saving time.

Imagine we want to parse the timestamp from our Postfix example line, i.e. `2020-04-04T06:59:43.000422+02:00` and use it for session timeout. As mentioned above, we need to be able to calculate the difference between two timestamps easily, and we also need to let the user specify the format of their timestamp. There is a C library function called `strptime`, which does most of the work we need. We can give it the textual timestamp and a format string and it will produce a Unix timestamp — the number of seconds since `1970-01-01T00:00:00Z` (also referred to as "seconds since the Epoch"). In our example, we would pass it `2020-04-04T06:59:43` as the timestamp and `%Y-%m-%dT%H:%M%S` as the format string and get something like `1585983583`.

However, since `strptime` works with whole seconds, we will need to implement sub-second precision ourselves. To achieve that, we represent our time as the number of microseconds since the Epoch (rather than entire seconds). The user can then supply the sub-second precision of the timestamp in a separate attribute. Also, although `strptime` works with time zones, it takes the time zone from what is set in the system, and we need to read the zone info from the messages themselves. To do that, we take a third attribute, which contains the time zone offset, and we subtract this offset from the timestamp manually.

In our example, the user would supply attributes containing `2020-04-04T06:59:43`, `000422` and `+02:00` and the format string written above. We would initially use UTC as the timezone and get exactly the value `1585983583` from `strptime`. Then we would subtract two hours from it obtaining 1585976383. Then we would handle the microseconds by multiplying the timestamp value by 10^6 and adding `000422`, which means we would get `1585976383000422` as the final timestamp.

The content of the attribute containing the time zone should have the format of sign, two digits representing hours followed by a character and two digits representing the minutes of the offset. The sign of the offset should correspond to RFC 5424 — e.g., if 2 hours need to be added to the current UTC time to obtain the local time, the offset should be +02:00. If the timezone attribute is not supplied, all timestamps are assumed to be in UTC.

The sub-second precision attribute should contain digits to be interpreted as being behind the decimal point in the number of seconds. So, for example, to represent time half a second after a whole second, this attribute can contain "5", "50" "500", . . . , "5000000". If this attribute is not supplied, zero is assumed.

2.4.2 Back to our example

So far in our example log, we've done some parsing which ensures that all desired messages arrive at one node (an option) of the tree and have `mail_id` set appropriately. We can put a call node under this option to process the fact that such a message was encountered. Since we parsed the timestamp, the called Python function will have access to the time the message was logged.

We create our own class *mail*, whose instances we will register as sessions identified by mail_id. We can store all log lines related to a particular e-mail in the associated *mail* instance will print these lines when the session is ended. Each time the *call* action is performed, we search for the session identified by `mail_id`. If it doesn't exist, we create and register it. Once we have the session, we can insert the input line in that session and either end it or leave it active and return.

Recall that in the tree we created for the example, analysis gets called when the option with pattern `%{mail_id=[0-9A-F]*}: %{_content=.*}` succeeded. This means that when it is called, the `_content` attribute is filled, and to detect when to end the session, we may look for _content being equal to `removed`.

If we don't set a timeout, then all sessions without an explicit `removed` message in the input log will be printed at the end of the output. We could print them after a certain time (say 30 minutes) of inactivity using the timeout feature.

The code may then look similar to the one below. This is just to give an idea; although it is very close to the real interface, you should refer to the documentation for the exact method names, arguments, and semantics.

```
class mail(session):
    def __init__(self):
        self.messages = []
    def end(self):
        for msg in self.messages:
            print(msg)

def process_msg:
    # find or create and register the session instance
```

```
instance = mail.get(
    context.mail_id,
    start_time=context.timestamp,
    timeout=24*60*60
)

# reporting activity on the session
# only needed when using timeout
mail_id.keep_alive(context.mail_id, context.timestamp)

instance.messages.append(context.msg)
if context._content == "removed":
    mail.terminate(context.msg_id)
```

3. User documentation

This chapter contains the complete user documentation to our program, which we named *Beaver*.

Welcome to Beaver user documentation. Beaver is a log analyzer targeted at Linux. Besides basic parsing using a tree structure, regular expressions, and storing attributes, it supports user programming in Python, reading from a Unix socket, and monitoring a directory for log files.

3.1 Installation

- Install the following prerequisites: libgrok, libpcre, python3, and python3-dev, more precisely the following worked for Ubuntu 20.04: `apt-get install libgrok-dev libpcre3-dev python3 python3-dev`. You will also need g++ supporting `-std=c++17` and make.
- Run `make install` as root. If you don't want to install Beaver to a system directory, you can just run `make` but keep in mind that you will need to set `PYTHONPATH=/path/to/beaver/repo/python`.
- If you have run `make install` and wish to remove the generated binaries from the current directory run `make clean`. Or you could remove the directory with the repository altogether.

If you wish to remove Beaver you can run `make remove`.

The following versions were used during development: python 3.8, gcc 9.3, libpcre3-dev 2:8.39-12. Beaver has also worked with python 3.7.

3.2 Basics

The simplest way of using Beaver is to let it read the log from the standard input and write the results to the standard output. In this case, only one argument needs to be specified — the name of the configuration file.

The input is read line by line and processed as specified by the configuration file. In a simple case, it contains a tree structure: the input line is matched against given regular expressions, and depending on which regular expressions match the line, the execution descends into different branches of the tree. Each successful match fills attributes associated with the input line. These attributes can be further matched. A simple tree could look like this:

```
match: msg { # attribute msg contains the original message
    {
        # if the content of msg matches this regexp, execute
        # actions enclosed in the brace pair
        # attribute err_msg gets assigned the part matching [^ ]*
        "error: %{err_msg=[^ ]*}"
        print: "error message encountered: {err_msg}"
    } # end of option "error: ..."
} # end of match of msg
```

This would filter lines containing `error:` and print the part after it together with a commentary `error message encountered:`. So `anything more_content error: nothing to say` would become `error message encountered: nothing to say`. You may therefore wish to write a more specific regexp, anchored at the beginning of the line.

A tree named `main` is located and then executed for every line of the log. Other trees may be defined in the same way and called using `parse_by: treename`. See the definition of the parse_by action for more details.

3.2.1 Tree structure

The tree consists of two basic kinds of nodes - actions and options with match being a special case of an action.

- Each `match` specifies what attribute should be matched and what options are its children.
- Each option contains a regular expression and specifies which actions are its children. If the selected attribute matches the option's expression, all of its children (all of which are actions) are performed (unless `break` is used — see below).
- The last kind of node with children is the root, which only specifies what actions are its children and if the tree is used, all of these actions are performed.

The `break` keyword allows you to break from a block if an attribute was successfully matched. It can be specified

- after an option — in that case, it indicates that if the match for this option succeeds all options under the same match node, which follow this option, should be skipped.
- after a match — in that case it indicates that if any of the options under it are matched successfully, the following actions under the same parent should be skipped.

As an example, consider the following tree:

```
match: msg {
    { "error"
        print: "error encountered: {msg}"
    }
    { "warning"
        print: "warning encountered: {msg}"
    }
} break
print: "looks ok"
```

Let us parse a few messages with that tree:

- `error: syntax error` — The match is performed, which retrieves the value of `msg` - the original message. The match of the first option succeeds and `error encountered: error: syntax error` gets printed. The second match does not succeed, so the second print is not executed. Then the control leaves the match. Because one of the options in that match was matched successfully, all other actions under the root of `main` are skipped, so the third print is not performed.

- `warning: return value not set` — The match again retrieves the value of the message, the first option does not succeed, but the second does and `warning encountered: warning: return value not set` gets printed. The break again ensures skipping of the third print.

- `build succeeded` — Neither of the options in the match succeeds, so the first two prints are not performed. When control leaves the match, it notices that none of its options were matched successfully and performs all the actions following the `break` keyword. Therefore `looks ok` gets printed.

- `error: the following warning is treated as an error` — The first option in the match succeeds and `error encountered: error: the following warning is treated as an error` gets printed. The second option is also matched successfully and `warning encountered: error: the following warning is treated as an error` gets printed. The break then skips other actions under the root, so the third print is not performed.

We might want to change the behavior in the last case and instead treat messages which contain both `error` and `warning` as errors only. To do that we would modify the tree by adding a `break` after the first option, this way:

```
match: msg {
    { "error"
        print: "error encountered: {msg}"
    } break
    { "warning"
        print: "warning encountered: {msg}"
    }
} break
print: "looks ok"
```

Now when the last message is parsed, the first option is matched successfully and its print is performed. The second option is skipped because the first one succeeded, and the third print is also skipped because one of the options under the match succeeded.

3.2.2 Patterns

Regular expression patterns are used in the match and option nodes. The syntax is that of the tool Grok (the patterns are actually internally passed to `libgrok` for compilation). The basics are the following:

- As mentioned in the Basics section, the beginning of the expression doesn't need to fit the beginning of the matched text, and the same holds for the end. This means that the regular expression `value` also matches the text `there is no value here`. Use `^` and `$` to anchor the expression to the beginning and the end, respectively.
- Round brackets by default have their special meaning (like in extended regular expressions of grep).
- Parts of the matched text can be named by enclosing the corresponding part of the expression into `%{}`, e.g., `%{attr_name=fo*}`.
- Named regular expressions (imported from a dedicated type of file) can be used with the following syntax: `%{my_expr:attr_name}` (note that in this case the attribute name comes second, unlike in the previous example)

3.2.3 Calling user-defined functions

The user can write his own Python scripts and call functions defined there using the `call` action, like `call: mymodule.myfunc`. The modules are searched for in the same directory where the configuration file lies and in directories specified by the `PYTHONPATH` environment variable. The functions should take no required arguments and their return value will be ignored.

Additionally, Python code can be written directly in the configuration file using the execute action like this:

```
execute {
    # Note: using Python's print is not recommended in most cases,
    # see below for more information.
    print("hello from Python")
}
```

Every piece of inline Python code is executed inside some Python module. You can specify which module should be used with the keyword `in`:

```
execute in mymodule {
    # Python code
}
```

3.2.4 Multiple trees

So far, we only saw configs with a single unnamed tree. You can also define other trees in the same configuration file by specifying their name and enclosing its body in curly braces, like so:

```
mylibtree {
    print: "hello from library tree"
}
```

These so called *library trees* can then be called from other trees using the `parse_by:` keyword. We will describe this in more detail in the List of supported actions section.

You can also explicitly name the main tree, so instead of

```
print: "hello world"
```

you could write

```
main {
    print: "hello world"
}
```

3.3 Actions

We already encountered the basic actions: match (in the Tree structure section), call and execute (in Calling user-defined functions section). Here we describe the remaining ones.

3.3.1 Timestamp parsing

The syntax is as follows:

```
timestamp: {
    format: "%Y-%m-%dT%H:%M%S"
    from: _attr_name
    micros_from: _micros
    zone_from: _timezone
}
```

The `format` argument needs to be a string literal containing the format of the timestamp compatible with `strptime`. It includes parts, which are matched literally (like the `T` in the format above) and some input field descriptors starting with `%`, which are expanded. See `man strptime` for details but some examples are:

- `%Y` denotes a complete four-digit year
- `%y` denotes a year within the century (only two digits)
- `%m` denotes a month number (e.g., 01 for January)
- `%d` denotes the number of a day within a month
- `%H`, `%M` and `%S` represent an hour number in 24-hour format and minute respectively

- `%T` is equivalent to `%H:%M:%S`
- `%I` represents an hour in 12-hour format

The `from` argument specifies the name of the attribute from which the timestamp should be obtained.

Only the `format` and `from` arguments are mandatory. There is no need to adhere to this particular order of arguments.

`micros_from` specifies the name of the attribute which should be parsed for sub-second precision of time. It should contain a number which is interpreted as being behind the decimal point (so for example if it has 3 digits, it is interpreted as milliseconds, if 6 then as microseconds).

`zone_from` specifies which attribute the timezone should be read from. When not specified, all timestamps are assumed to be in UTC. The attribute value should be in the format plus or minus (optional) immediately followed by the number of hours, one character (any character) and number of minutes specifying the offset from UTC (so for example Central European Time would be `+1:00`). If the number of minutes is missing, 0 is assumed.

3.3.2 Print

The basic syntax is as follows:

```
print: "attribute foo contains {foo}"
```

The parts in braces are either names of attributes, which should be printed, or escapes.

Escape sequences are:

- {`&quote`} for `"`
- {`&lbrace`} and {`&rbrace`} for { and } respectively

It is also possible to print the whole context — all non-empty attributes whose names do not begin with and underscore, which prints context's attributes whose name doesn't begin with an underscore.

The attributes are printed one per line in the format *attr_name*: *attr_value*, followed by an empty line.

The syntax for context print is

```
print: context
```

To better handle situations where a binary dump is a part of the log (or due to some bug, the message contains strange characters), all characters with ASCII code smaller than 32 or larger than 126 are escaped using backslash and three decimal digits representing their code (there are always exactly three digits). This means that non-ASCII UTF-8 characters are escaped. This behavior might change in the future. The digits may also change to hexadecimal.

3.3.3 parse_by

`parse_by:` keyword indicates that parsing using a named library tree should be done. The library tree may be in the same file before or after the tree using it or in a different file. The syntax is either `parse_by: libtree`, in which case the tree is searched for in the current file, or `parse_by: libfile:libtree`, in which case the configuration file `libfile` is parsed and searched for the tree.

Instead of the name `libfile`, you could specify a path to the file. If the path is relative, it is considered relative to the directory containing the main config (the config you gave to Beaver as a command-line argument). This behavior can be changed by setting the `BEAVER_LIBPATH` environment variable to a path to the desired directory. You can also use absolute path to the library file.

3.3.4 Default module

`default_module` specifies what should be the default Python module to look for a function referred to by a call action. Inline Python is also by default executed in the default module. Function names containing dots and inline Python with explicit module specification (using the `in` keyword) are not affected.

The `default_module` is scoped lexically to the subtree under its parent node. This means it will only affect nodes in this subtree, but it may not even affect all of them since it is performed at the same moment as any action. Therefore actions preceding it under its parent node will not be affected. For example, imagine you are writing an option and specify `default_module` under it like so:

```
{ "regexp" # default_module will be scoped to this subtree
    # some preceding actions
    call: foo
    match: attr {
        { "error"
            call: bar
        }
    }

    default_module: my_module

    call: f
    match: attr_2 {
        { "warning"
            call: process_warning
            parse_by: libtree
            call: different_module.func
        }
    }
}
```

Then the calls of `foo` and `bar` will not be affected by the `default_module: my_module` while the calls to `f` and `g` will. The call to `different_module.func` contains a dot, so the `default_module` doesn't affect it, and `func` is looked for in the module named `different_module`.

What's more, trees called from the subtree where `default_module` takes effect, will not be affected. So, in this case, `libtree` would not know about the `default_module: my_module` specification.

3.3.5 List of supported actions

The supported actions include:

- root and match — discussed in the Tree structure section
- call and execute — see the Calling user-defined functions section
- timestamp parsing — refer to its dedicated section (with the same name)
- print — see the Print section
- `parse_by` — see the parse_by section
- `default_module:` — see the Default module section
- `import_patterns`: — Imports named patterns from the specified file for use in the regular expressions. Their scope is the same as that of `default_module`.

3.4 Python interface

Both functions from scripts and inline Python (the execute action) can use the context object by referring to it as `c` and accessing its attributes simply by dot syntax, like `print(c.msg)`. Attributes can also be assigned from Python (and even new attributes created) by writing `c.my_attr = "value"`.

All of the attributes are of type `str` except for `c.timestamp`, which contains the number of microseconds since `1970-01-01T00:00:00 UTC`. That one is of type `int` and cannot be assigned to.

In case the attribute name is only known at runtime and stored in a variable `name`, you can use `getattr(c, name)` and `setattr(c, name, "this is the value)`. Do not use `c.__getattr__` or `c.__setattr__` as they don't work (unless you're looking for an attribute named `__getattr__` or `__setattr__` in the context).

The interface for other activities is in a Python module named `b` (short for Beaver). It is imported inside the session module (described below) so if you use sessions, there should be no need to do it explicitly.

The program buffers its output before printing it. Since this buffer is different from that of Python, it is not recommended to use the default Python `print` because the printed message could appear at a very different place in the output than where it was intended to be. To address this issue, Beaver's Python interface contains a `print` function, which handles output buffering in a way compatible with the rest of Beaver's output. It does not add a newline at the end. To achieve that, `println` should be called, which accepts an optional single argument. The arguments of `print` and `println` are converted to strings as if the built-in `str` function was called on them. In case you need to print text with variable substitutions, we recommend using Python f-strings, e.g., `b.println(f"attribute` `my_attr contains {c.my_attr}")`. Also, please note that strange characters are escaped in the same way as with the print action.

Scripts may also call a parsing tree using the `b.parse_by` function and passing to it the name of the tree including the file name. This is the same as in the `parse_by` action described above except that the name must include the file name and it doesn't guarantee parsing of that configuration file (so it needs to be mentioned in another configuration file or be the main file).

Warning: When dealing with time, please note that the current timezone will always be set to UTC. See the Tips, tricks and caveats section for more details.

Closing curly braces — } — may be a part of the inline code as long as they are indented by at least as many whitespaces as the first line of the code.

3.4.1 Sessions

Session support is intended for handling objects like ssh sessions. It can, however, be used in a rather general way. A session is an instance of a class inheriting from `session` (defined in the `session` module). The user can register a session instance by using class method `session.register`. A session may have a *timeout* set after which — if `keep_alive` method is not called — the session is ended.

Sessions are identified by a string and their class. We recommend creating your own class inheriting from the supplied `session` class and register instances of that derived class. If this class has an instance method `end`, it is called when the session is ended - one string argument is given, which indicated the reason for ending the session. This `reason` argument is one of:

- `manual` — when `session.terminate` is called
- `timeout` — when the session timed out
- `end` — when the program is ending, all sessions are ended with this reason given to them

The `session` class has the following class methods:

- `find(cls, ident)` — Finds the session identified by `ident`. If there is no such session, it returns `None`.
- `get(ident, *args, start_time=0, timeout=0, **kwargs)` — Tries to find the session identified by `ident`. If there is no such session, it gets created. `args` and `kwargs` are passed to the `__init__` method. If `timeout` is not 0 the session's timeout interval is set to `timeout`. If `timeout` is 0, the session is never timed out.
- `terminate(ident)` — Ends the corresponding session calling its `end` method passing `manual` as the reason for termination.
- `keep_alive(ident,reference_time)` — Indicates that the timeout of the session should be postponed to `reference_time` + `timeout` (where timeout is the argument that was passed to `get` or `register`). Please note that sessions are timed out after the complete processing of each message. Imagine

that you have a session that times out at time T. When processing the first message with timestamp larger than T, the session will still be alive and if `keep_alive` is called, it can still be kept.

There is also one instance method of `session`:

- `register(ident, start_time, timeout)` — Registers an existing instance. This means Beaver will store it, consider it a candidate for a session to time out and end it if Beaver itself is terminating.

3.5 Command-line arguments

In its simplest variant, Beaver can be run with just the configuration file name as a single command-line argument, e.g. `beaver example.conf`.

These command-line arguments are the supported:

- `--debug` — turns on debugging prints for both configuration parsing and input processing
- `--monitor` — turns on the monitoring feature: when the end of input is encountered, continue reading indefinitely so that new lines added to the file will be processed too. This flag overrides this setting from the configuration file - useful mostly for reading input from files in a directory (see `Reading from a directory`).
- `--no-monitor` — forcefully turns off the monitoring feature
- `--clean-run` — when reading files from a directory, progress storage can be used, which makes Beaver continue reading where it previously left off (see the section Reading from a directory for more details). This option makes Beaver behave as if it were empty

3.6 Examples

Imagine we're processing a Postfix log whose lines have approximately the following format:

```
2020-04-04T06:59:45.724399+02:00 postfix/qmgr[15441]:
 671641C38D9: removed
```

We would like to count the number of e-mails sent/received every minute. For detection of one message, we use log lines with the e-mail identifier and a message stating `removed`. We can write the configuration below. Please note that the line break in the first pattern is purely typographical — it should be written as a single line in the config file.

```
default_module: counter
match: msg {
     { "%{time=[[:digit:]-]*T[^+.]*}\.%{micros=[^+]*}\+
%{offset=[^ ]*} %{_message=.*}"
        timestamp: {
            format: "%Y-%m-%dT%H:%M:%S"
            from: time
            micros_from: micros
            zone_from: offset
        }
        match: _message {
            { "[[:alnum:]]+: removed$"
                call: inccounter
            }
        } break
        call: update_time
    }
}
```

We also create the corresponding Python script named `counter.py` (this name match the module name specified in the config), which we place in the same directory as the config.

Let's discuss a couple options of how to write the script. The first one will have a simple counter and store the current minute. That would look like this:

```
counter = 0
current_minute = None

def inccounter():
    global counter
    update_time()
    counter += 1

def update_time():
    global counter
    global current_minute
    t = c.timestamp
    t = t // 10**6
    t -= t%60
    if current_minute is None: current_minute = t
    while current_minute < t:
        b.println(f"{current_minute} {counter}")
        counter = 0
        current_minute += 60
```

The problem with this is that it will not write the number of e-mails for the last minute in the log. We could instead have a session for each minute, which would take care of counting the number of e-mails in it and print the number of e-mails in its `end` method. We can then register the session with a timeout of one minute, which will ensure it is ended at the end of the minute or when Beaver itself is terminating. This approach would yield the following script:

```
from session import *

class mail_counter(session):
    minute = 0

    def __init__(self, timestamp):
        self.counter = 0

        # the timestamp is in microseconds, so
        #timestamp % (10**6*60) is the number of microseconds
        # since the current minute started
        timestamp_minute = timestamp - (timestamp % (10**6*60))

        self.beginning = timestamp_minute // 10**6
        if mail_counter.minute < timestamp_minute:
            mail_counter.minute = timestamp_minute

    def end(self, reason):
        b.println(f"{self.beginning} {self.counter}")

    def update_time():
        """ register all mail_counter instances we need
            (there might be no messages in some minutes)
        """
        t = c.timestamp
        t -= t%(10**6*60)
        if mail_counter.minute == 0:
            mail_counter.minute = t - 10**6*60
        while mail_counter.minute < t:
            t2 = mail_counter.minute + 10**6*60
            mail_counter.get(
                str(t2),
                timestamp=t2,
                start_time=t2,
                timeout=60
            )

def inccounter():
    t = c.timestamp
    t -= t%(10**6*60)
    update_time()
    current = mail_counter.find(str(t))

    assert current is not None
    current.counter += 1

def update_time():
    mail_counter.update_time()
```

However, this script might seem rather long for the task it performs. Luckily, there is also a third option, which stems from the first script but doesn't suffer from the same flaw. We will use a part of a trick described in the Tips, tricks and caveats section. We will register a session, which will be ended when Beaver wants to terminate. This session can print the counter in its `end` method. The code would look like this:

```
from session import *

counter = 0
current_minute = None

def inccounter():
    global counter
    update_time()
    counter += 1

def print_counter():
    global counter
    global current_minute
    b.println(f"{current_minute} {counter}")
    counter = 0
    current_minute += 60

def update_time():
    global current_minute
    t = c.timestamp
    t = t // 10**6
    t -= t%60
    if current_minute is None: current_minute = t
    while current_minute < t: print_counter()

class end(session):
    def end(self, reason):
        print_counter()

end.get("", timeout=0)
```

We can now run `beaver counter.conf <input.log`.

3.7 Other ways of reading input

3.7.1 Directory

Besides reading from stdin, it is possible to read from all the files which are located in a specified directory and whose name matches a regular expression. To achieve that, write the following in the config:

```
input: {
    path: ./input-[[:digit:]]\.in
    sort: num
    progress_storage: /path/to/progress_storage
    monitor
}
```

The first two arguments — `path` and `sort` — are required. The `path` is interpreted as literal path to the directory followed by a slash and then a regular expression that does not contain spaces, slashes or hashes. The slash must be present even when reading from the current directory.

All files whose whole name matches the regular expression (e.g., the expression `bar` is equivalent to `^bar$` and therefore doesn't process a file named `barbarian`) are processed in the order specified by `sort`. The possible orders are the following:

- `lex` - The file names are sorted lexicographically
- `num` - The file names are sorted numerically by the largest suffix that is a (integer decimal) number. If no suffix is a number, the file is treated as having a 0 at the end. This is useful for rotating logs. The order of files having the same number is currently undefined. It may be later changed so that files having the same number are sorted lexicographically.

Optionally a file for *progress storage* may be specified. In that case, when the program terminates, it saves for each file it processed the information how far it got. When the program is later run again with the same progress storage, it continues where it stopped in each of those files (if the files haven't grown, they are not processed again). This behavior can be changed by specifying `--clean-run`, in which case Beaver ignores the content of progress storage when it is starting (but writes into the storage in the same way it does without `--clean-run`). The files are identified by inodes, and therefore they are still treated as the same file even if they are renamed.

The optional `monitor` argument specifies that the execution should not stop after all of the files found are processed, but instead, Beaver should keep running and wait for some of the files to grow or more files to appear. In such case, the program should be stopped using SIGINT or SIGTERM.

3.7.2 Socket

The program can also open a Unix socket and read messages from it. Unix sockets are sockets for local interprocess communication, which may or may not be bound to a filesystem pathname. In the case of Beaver, they are bound to the pathname you specify.

The socket is configured by:

```
open_socket: /path/to/socket
```

When reading from a socket, it is expected that the messages will comply with the syslog protocol in starting with <*priority_value*> (e.g. <`13`>). This prefix is removed from the message (to enable paring with a tree which can also be used for processing log files) and the number is split into facility and severity as per the syslog protocol and these numbers are assigned to attributes names `facility` and `severity` (which are left empty when not reading from a socket).

If the prefix of the message isn't in the form <number> then no prefix is extracted, and the `facility` and `severity` attributes are left empty.

3.8 More on configuration file syntax

This section contains a more detailed and somewhat more formal description of the configuration file syntax.

The configuration file consists of words separated by whitespaces { or } (which are considered to be words of one character) and string literals, which are delimited by quotation marks. The number of whitespaces between words plays no role as long as it is at least one and it's not inside quotes or around inline Python. If newline is present inside a string literal, it is included in the string.

Words and curly braces create directives and blocks. A directive is a keyword followed by a fixed number of arguments, e.g., `parse_by: libtree`. Arguments can be words or string literals, depending on the keyword used. It is recommended but not required to specify the keyword and all of the arguments on the same line and end the line after the last argument.

Blocks begin with { and end with }. They may be preceded by a keyword and possibly arguments based on the particular block type. One block may be written within another. Supported blocks are:

- tree — optionally preceded by tree name, if no name is specified, `main` is assumed
- match — must be preceded by `match:` keyword and the name of the matched attribute
- option
- inline Python — preceded by the `execute` keyword, possibly followed by the `in` keyword and the name of the module to execute the code in
- input — preceded by the `input:` keyword
- timestamp parsing — preceded by the `timestamp:` keyword

The possible location and content of these blocks stems from their semantics:

- Trees must be specified as top-level entities of the configuration file, not within any other block. And since each root has actions as its children, it can contain actions.
- a match is an action and therefore must be specified within an option or as top-level action in a tree. It contains options.
- option node is always a child of a match node, so its block must be within a match block. Each option block must contain a string literal at its beginning. This string is interpreted as the regular expression of that option. It can then be followed by actions.

- inline Python is an action, so it can be present in the same places as match and it can contain only Python code.
- input is a top-level entity, just like trees. It is recommended but not required that if you specify it, you do so at the start of the configuration file.

Comments are also supported - they start with `#` and continue to the end of the line. The `#` character does not start a comment when inside a string literal. The character also loses meaning in inline Python — so, for example, it can be part of a Python string literal.

Ends of line play the same role as any other whitespace except:

- They terminate comments
- They play a role around inline Python. The rest of the line after the opening brace is still treated as a part of Beaver config and can only contain comments. The Python code needs to begin on a new line. Inside the code, they play the same role as in a regular Python code. The closing brace must be indented by fewer whitespaces than the first code line

3.9 Tips, tricks and caveats

3.9.1 Output buffering

The output is buffered. When reading from a directory, the output is flushed every time the end of a file is encountered. However, if you read input from a socket or standard input and there is currently no input to process, but the program isn't terminating, you may not see the whole output.

3.9.2 Explicit importing of trees

Calling a tree from Python using `b.parse_by` does not guarantee that the appropriate configuration file will be parsed. If needed, you should define a dedicated tree in the configuration file and put a `parse_by` action of the missing tree in it. For example, imagine have `libtree` in `libtree.conf` but you use `beaver main.conf` and only call `libtree` from Python. Then you can put this in your `main.conf`:

```
import {
parse_by: libtree.conf:libtree
}
```

The block named `import` is just a regular tree, which gets never used, but it ensures parsing of `libtree.conf`.

3.9.3 Message-independent variables

Since modules are only imported once, their global variables outlive individual calls of their functions. So, for example, to implement a function counting how many times it was called, you could write the following:

```
import b

counter = 0

def f():
    global counter
    b.println(counter)
    counter += 1
```

Similarly, if the module above is called `module`, you can increment and print the counter in inline Python instead of calling `f` like so:

```
execute in module {
    global counter
    b.println(counter)
    counter += 1
}
```

3.9.4 Initializing and finalizing Python modules

All functions called from the config get invoked every time their parse_by node is encountered. You may instead wish to initialize some variables once before parsing the first log message. You may also want to run some piece of code after processing the last message.

To do the former, you can just write the code you need executed directly inside a Python module as if you wanted it executed when calling `python3 module`. To accomplish the latter, you may register a session with no timeout and place the code in its end method. The following is an example of a module, which does both:

```
from session import session
import b

class global_session(session):
    def end(self,reason):
        b.println("module deleted")

b.println("module imported")
global_session.get("")
```

3.9.5 Timezone in Python

Because Beaver handles time zones manually, it sets the `TZ` variable to `UTC+0` to prevent `stat`. It seems that without setting it, the C time operations are slow (apparently `/etc/localtime` is `stat`-ed every time). This speeds up the execution but affects Python too, so the timezone will always be set to UTC.

Imagine for example that the timestamp in the current message was parsed with `zone_from` specified and the appropriate attribute contained `+01:00`. When you call `datetime.datetime.fromtimestamp(c.timestamp).isoformat()`, you will get a timestamp that is one hour behind the one in the log. Please note however that even if the timezone was kept to the system defaults, you would still have this problem when processing logs whose timezone differs from yours (even if the difference was just daylight saving time).

There are plans to add a built-in attribute specifying the offset from UTC in minutes (calculated from the attribute supplied with `zone_from:` when parsing timestamp). However, the current implementation lacks this feature.

3.9.6 Recursive parsing

Imagine you would like to parse any number of key-value pairs that appear in an attribute and for each pair create a new attribute whose name is the key and the value is the value from the pair. This could be desirable, for example, to parse the structured data element of RFC 5424.

You cannot write one match that would assign all the necessary attributes, but you don't have to leave all the work to Python either. You can write a tree that detects the first pair, assigns its name and value to the appropriate attributes, uses one line of Python to do the assignment we want, and then recurses.

This is a config that parses space-delimited key-value pairs directly from the input and then prints the context:

```
parse_by: key_value
print: context

key_value {
    match: msg {
        # finds out the first "somekey=somevalue" occurrence
        { "^%{_key=[^ =]*}=%{_value=[^ =]*} *%{msg=.*}"
            execute {
                setattr(c, c._key, c._value)
            }
            parse_by: key_value
        }
    }
}
```

However, please note that this will have quadratic time complexity, so it will probably be faster to do the parsing manually if there are many key-value pairs in one message.

Also please be aware that Beaver does no tail recursion optimization and instead recurses itself. What's more, there is currently no limit set on the depth of the recursion, so infinite recursion will cause Beaver to crash with segmentation fault.

3.9.7 Performance differences among patterns

Please be aware that some patterns, which are similar and sometimes even equivalent, may give very different performance. For example, imagine we are filtering messages containing `removed` preceded with at least one character. If we use `.removed` as the pattern, we got much faster execution than when we used `.+removed`, by a couple of orders of magnitude.

This could be problematic if you need to assign the part matching `.+` to an attribute. It could help to filter the messages first by the faster pattern, and only then do the attribute assignment, like so:

```
match: msg {
    { ".removed"
        match: msg {
            { "%{myattr=.+}removed"
                print: "{myattr}"
            }
        }
    }
}
```

3.9.8 Error reporting

When a syntax error is reported, its position is reported, which consists of line number and character number within that line. The character number refers to the character where Beaver's parser first identified the error. Since the parser often reads the config by words, this position will likely be the end of a word. So, for example, if you write `print: foo` instead of `print: "foo"`, the character number will refer to the second `o` in the word `foo`. The error message will state that a quotation mark was expected but `foo` was found.

If you misspell the name of some action, the error message will state that a closing brace was expected (which would end the block) but your misspelled keyword was found. A similar message is printed, for example, if you specify an action where Beaver expects an option. Use the file position reported to figure out what happened.

These caveats may be mitigated in the future.

4. Programmer documentation

This chapter contains Beaver's complete programmer documentation.

Beaver is written in C++. It uses libgrok for pattern matching and Python C API for interaction with user scripts and inline Python.

A high-level overview of a typical program run is as follows:

- Configuration is parsed first, which builds a tree with options and actions. This tree is represented by its nodes, each of which remembers its children. If there is some Python involved, its modules are imported at this point.
- Input initialization is handled — a file is potentially open, a directory is potentially listed. If progress storage is enabled, it gets loaded.
- A context is created and the input is read line by line. For each line, the tree is used for parsing (each action node has a `perform` method implementing its behavior and calling `perform` of nodes under it if appropriate). The context gets re-initialized (attribute values get cleaned) and re-used between the lines of input.
- When the end of input is encountered, then if the monitor option is set, the program waits for more input or a SIGINT or SIGTERM signal.
- If there are sessions involved, they are all terminated. Then Python's `Py_Finalize` gets called. Also, if progress storage is enabled, the progress is saved.

4.1 Tree representation

The tree nodes are instances of either `struct MatchOption` or a class derived from `ParseAction`. Each node holds pointers to its children (standard pointers in case the children are actions and unique pointers in case the children are options). Actions implement a method `perform`, which is called when the action should be done and also takes care of performing all appropriate actions in the subtree under it. Options, on the other hand, are just a passive data structure. The only actions that have children are:

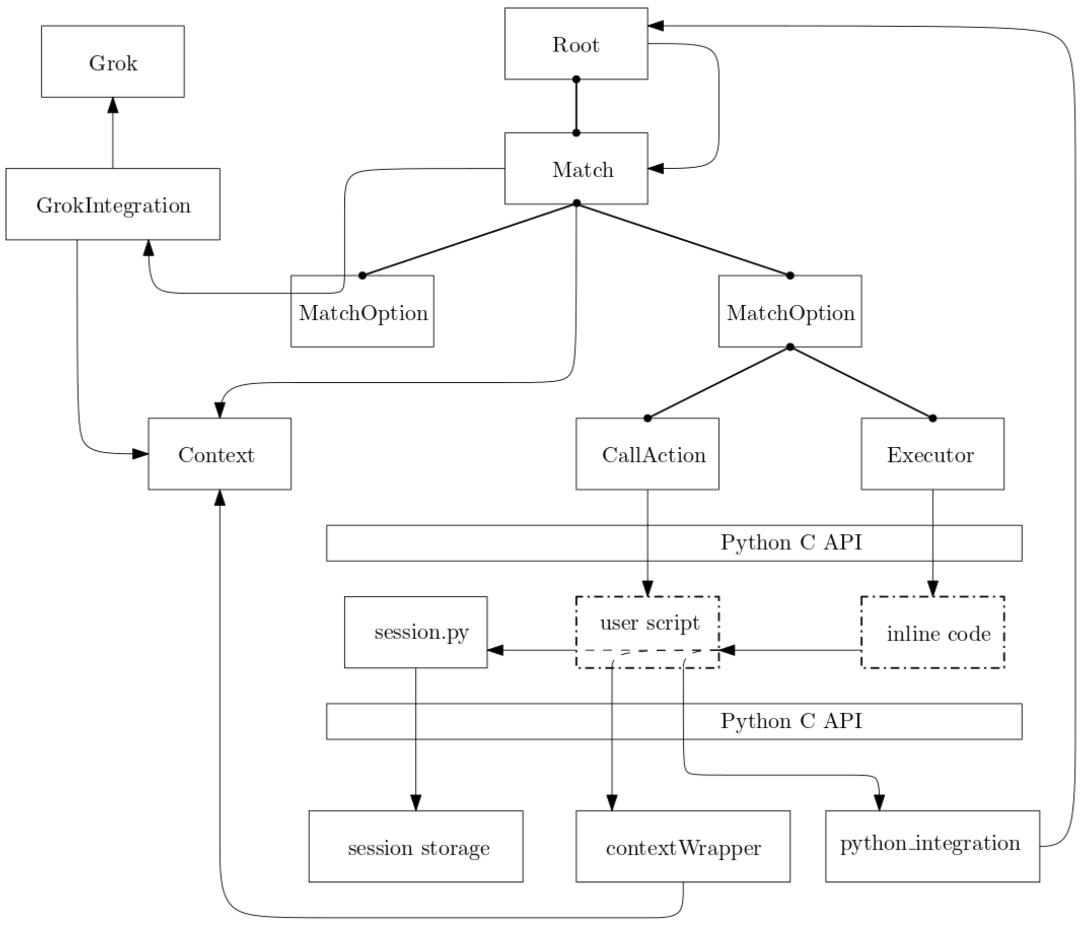
- `Root`, which stores a vector of actions (its children) and its `perform` simply calls them one by one
- `Match`, which has options as its children (also stored in a vector) and its `perform` iterates throw them. For each option, it tries to match the desired attribute's content against the pattern of the option. If the match is successful for a certain option, call it `opt`, the `perform` method iterates through all actions stored in `opt` and calls perform on them before continuing to the next option.

The tree structure is only stored by nodes representing their children and the tree is held as a pointer to its root. To perform parsing using the tree, we only need to call `perform` on its root.

Besides nodes calling `perform` on their children, some nodes may call `perform` on other tree's root to implement `parse_by`. This is why options store regular pointers to actions and not unique pointers — roots can be pointed to by multiple locations. Shared pointers are also not a great choice because they could create cycles (tree *A* calling tree *B*, which calls tree *A*). Actions are therefore stored separately in a static member of the `ParseAction` class named `_actionOwner`, where they are accessible and they could be deleted if needed. However, with the current implementation, all actions parsed remain valid until the program terminates and therefore are not deleted.

Trees themselves are stored inside `class TreeStorage` in its `_namedTrees` static member.

4.2 Overview of the components



The picture is an overview of components acting at run time (when processing input) and how they interact. The thick lines without arrows represent tree edges. Lines with arrows represent one component calling another.

The `Root` instance in the picture calls its children. The `Match` instance fetches the matched attribute content from a `Context` instance, and it calls the `GrokIntegration` stored in each of its children. This `GrokIntegration` communicates with Grok, and if the match is successful, it fills the appropriate attributes by calling methods of the `Context` class.

Each MatchOption has actions under it. The CallAction goes through the Python C API, and it calls the appropriate function in a script supplied by the user. This script has access to the session implementation written in Python. It also has access to a contextWrapper instance (under the name c) and python_integration (which is the implementation of the b module), both through the Python C API. The session class calls methods for session storage through the Python C API as well.

The ContextWrapper instance accesses and modifies attributes stored in the Context instance. Session storage manages sessions for the user, but it also gets notified when the program is terminating to end all sessions. The python_integration component calls the Beaver print function and calls other trees.

Another action associated with Python is the inline Python, implemented by the Executor class. An Executor instance calls the inline code, which is executed in the context of a user script. Therefor, the code can access globals of this script, including the contextWrapper instance, the b module (if it is imported), and the session module (if it is imported).

4.3 Attribute storage

Attributes are stored in what is called a context. Most attributes are kept in an array and the Context class contains a mapping of attribute names to their indexes (this is not completely precise, see below for intermediate vs. final attributes). The number of attributes and the mapping is found when processing the configuration file and matches reference attributes by indexes instead of their names. This is to speed up the process of attribute value lookup as this way, there is no need to search in an associative container.

The attributes are actually divided into *intermediate* and *final* ones, separated by naming convention - intermediate attributes' names begin with underscore. The semantic is that intermediate attributes are implementation details, while final attributes are to be considered our output. The original idea was that intermediate attributes would be thrown away after each line, while final attributes could be kept. However, there currently seems to be no need to keep even the final attributes, so the sole difference is that print: context only prints final attributes.

The attribute values themselves are stored as instances of struct str, which stores a char *ptr and the attribute's length. The strings pointed to by str::ptr aren't null-terminated and are located inside the original message (which lies in the buffer used for reading the input) to prevent the need for copying. (Initially, attributes were represented using std::string, but this turned out to be very slow - although probably mostly due to time needed for memory allocation; no testing was performed of having pre-allocated memory and copying the strings there).

The mapping of names to indexes is represented by a std::map<str, int> for each of the attribute kind (intermediate and final) - named _finalAttributeIndexes and _intermediateAttributeIndexes. The strs are stored in an array for each attribute kind; these are named _finalAttributes and _intermediateAttributes. In the end, we decided to create only one Context instance and re-use it for each line, but the original design was to allow multiple instances with the same set of attributes (and name to index mapping). Therefore, `_finalResultIndexes` and `_intermediateResultIndexes` are static members of the `Context` class. The attribute values, on the other hand, are linked to a `Context` instance, and therefore the `str` arrays `_finalResultIndexes` and `_intermediateResultIndexes` are ordinary members of the class `Context`.

To allow for attribute values to be set by Python, the `Context` needs to have a way of owning the stored attribute value (as `strs` only store a regular pointer, which does not guarantee the memory it points to will not be freed). To enable this, we introduced a container named `_attributeOwner`. `std::list` was used to prevent moving of these strings (it seemed during the development that although the pointers were to the first char and not the `std::string` object, the moving of these strings sometimes invalidated the pointer).

The user is also allowed to create new attributes from Python (by assigning to an attribute that does not exist yet). As these dynamically added attributes are linked to a single instance of `Context`, their name to index mapping is not stored in one of the static maps (which are common to all instances) but in an instance map `_additionalAttributes`.

Since dynamically added attributes are not expected to be there in every case, their containers are only created when they are needed. To be optional, they are stored as `unique_ptrs` making their final type `std::unique_ptr<std::list<std::string>> _atributeOwner` and `std::unique_ptr<std::map<str, str>> _additionalAtributes`.

4.4 Regular expressions matching

The libgrok library (which internally uses PCRE) is used for matching and interfaced with the main program through a dedicated class `GrokIntegration`. Each pattern is compiled as a Grok pattern using `grok_compile` during configuration parsing (in the constructor of `GrokIntegration`). During runtime, `GrokIntegration::tryMatch` is called, which calls `grok_match` and performs attribute value extraction.

Initially, the extraction was done using `grok_walk_init` and `grok_walk_next` but this approach turned out to be slow, so later the approach was changed. In the constructor of `GrokIntegration`, `grok_capture_walk_init` and `grok_capture_walk_next` are used to put `grok`'s captures into a vector called `_captures`. These captures contain their name, `pcre_capture_vector` (which is an array of pointers to beginnings and ends of matched string), and `pcre_capture_number` (from which the index into `pcre_capture_vector` can be easily computed).

In the destructor of `GrokIntegration`, `grok_free` is called.

As a part of the attribute value extraction, `GrokIntegration` fills the `context` with attributes inside its `tryMatch` method if the match was successful. The original approach was to use `GrokIntegration` purely as an integration layer between the project and Grok, so `tryMatch` took a lambda function for processing the attributes. Since there was only one lambda used, this was changed in an attempt to speed up the processing. Although the results don't seem significant, it was not changed back.

4.5 I/O

The initial naive approach was to use `std::getline` for input and `std::cout` for output. This (especially the `std::getline` part) turned out to be very slow.

The input is read into a buffer using the `read` syscall and buffered manually for speed. Then the buffer content is processed by looking for newline characters in it and parsing the line found each time this character is found. When a line longer than the buffer (4096 characters) is encountered, a warning is printed and the line is skipped. The reading and segmentation into lines is done inside `class Reader`.

The output is also buffered manually. To support printing warnings and debugging messages, variadic template functions `DebugPrinter::print` and `Warning::print` were written. As mentioned in the user documentation, some characters are escaped by a backslash and three decimal digits representing their code. For this reason, the writing into the buffer isn't done using `strcpy` or `memcpy` but a for loop. This is done inside the `print(const char *s, int length, bool escape = true)` function (where `escape=false` turns the escaping off).

`print` actions from the parsing tree are implemented by `Print` and `ContextPrint` classes, which inherit from `ParseAction`.

4.5.1 Directory monitoring

This feature is supported by `class DirMonitor`. The directory is listed using `opendir` and `readdir` calls, and the matching is done using `Grok` (`^` is added to the beginning of the pattern and `$` to the end since this better corresponds to the intuitive behavior).

When the program is started, it lists the contents of the directory, filters them by the pattern and then looks to `progress_storage` file (if it's supplied) for offset corresponding to each particular file's inode. Inodes are used because in a typical log rotation case the file names change, and therefore a file name is not a reliable identifier of the file itself.

The file names are sorted in the order specified by the user and opened one by one. Each time a file is, open Beaver seeks to the offset corresponding to it (0 if it hasn't read the file) and attempts to read. When the file's end is reached, it gets closed and another one gets opened. If all files have been processed and the `monitor` option is turned on, the program sleeps for a short while and repeats the process from the step of listing the directory onwards.

In the end, if `progress_storage` is specified, inodes from persistent storage for which a corresponding file (with its name matching the pattern) has not been found are forgotten. The rest of the inodes are written to the storage. To allow for this, a signal handler for `SIGINT` and `SIGTERM` is used (which also ends all sessions and does further cleanup by calling a function called `finalize`).

4.5.2 Socket

Standard `socket` and `bind` syscalls are used with `AF_UNIX` socket family. For removing the priority value from the message, only the `ptr` (the pointer to the beginning) of `msg` is incremented. Besides the incrementation of `ptr`, the input is read in almost the same way as it is read from files or from the standard input.

4.6 Python

Python C API is used for the interface between Python and C++. Modules are imported at the time of configuration parsing by calling `PyModule_ImportModule` and encapsulated by instances of `class CallAction`, which inherits from `ParseAction` and gets stored in parting trees.

In each module, the identifier `c` is set to refer to an instance of `struct contextWrapper`, which is a simple `PyObject` holding a C pointer to the active `Context` and implements methods for attribute access. These are equivalents of `__getattr__` and `__setattr__` in that they can be called like `print(c.attrname)` and `c.attrname = "attrvalue"` or use them by calling `getattr(c, "attrname")`, `setattr(c,"attrname", "attrvalue")`. Unfortunately, they currently cannot be called as `c.__getattribute__("attrname")` or `c.__setattribute__("attrname", "attrvalue")` because to do that, the `getAttribute` implementation would have to return some Python function object when invoked with name `__getattribute__` or `__setattribute__`.

Imported modules are cached in `static std::map<std::string, PyObject *> CallAction::modules`, since they can be shared by multiple `call` and `execute` actions.

4.6.1 Inline Python

inline Python is read from the configuration file. The base indentation is determined by the indentation of the first line following the opening brace. This base indentation (the same number of whitespaces regardless of their precise value, e.g., a tab is treated the same as a space) is then removed from all the code lines. The end of the Python code is detected when a closing brace with a lower indentation is encountered. If an indentation lower than the base occurs and the first (non-space) character is not a closing brace, an error is reported.

The code is then passed to `Py_CompileString` before the reading of input begins. It is then encapsulated in an instance of `class Executor` (which inherits from `ParseAction`). Each inline code block is executed in the context of some module (either specified by `execute in modulename` or by the default module). This is done by getting a reference to the module's globals when the `Executor` is created and passing it to `PyEval_EvalCode` each time the block is run. The only way the inline code can access the `context` or rest of Beaver is through this module.

4.6.2 Python integration

Since Python needs to call some functions built-in to Beaver (like those for session handling), the `b` module was introduced and is added to all user modules by calling `PyModule_AddObject`. On the C++ side, it is referred to as `python_integration`, and its code is in `python_integration.cpp`. It offers a `print` function (which calls Beaver's `print`), functions for handling sessions (`register`, `end`, `find`, and keep_alive), and a function call_tree. These are the implementation functions. There also exists a module session written in Python, which offers a base session class. This class offers class (and instance) methods as wrappers for these implementation functions. It also offers an additional method get, which attempts to find the session, and if the search is unsuccessful, it creates and registers it.

4.6.3 Session storage

As mentioned in the user documentation, sessions are stored using both a string identifier and their class as a key. They are stored in a std::map<std::pair<PyObject *, std::string>, session> named registeredSessions, where struct session stores a pointer to the registered session instance along with other metadata for supporting timeout. This includes the time when the session should time out, by how much this time should be extended when keep_alive gets called, and two iterators, which we will discuss now.

Very often (ideally after each message), sessions that expired need to be ended. Going through all registered sessions every time would likely be time-consuming, so sessions also need to be stored by time in a way that will enable quickly identifying such sessions (as well as quickly inserting them when they get registered). This is done in std::map<timeType,session *> sessionsByTime. However, sessions can also be ended manually (which is expected to be the usual case)wh, and therefore, when ending a session this way, it needs to be (quickly) removed from sessionsByTime. Sessions also need to have their timeout changed when keep_alive gets called. Our data structure needs to be able to adapt to this quickly.

To support this, struct session also contains an iterator into registeredSessions pointing to itself and a similar iterator to sessionsByTime. When a session is ended manually, the iterator into sessionsByTime is used for removing the session from this map. When a session times out, the iterator into registeredSessions is used. For the actual identification of sessions, which should be timed out, the project walks through sessionsByTime, ending sessions encountered until it finds a session that is yet to live.

Sessions are timed out after the complete processing of each line of the log. It cannot be done earlier because the timestamp is parsed as a part of the parsing tree.

4.7 Configuration parsing

Configuration parsing is done manually by a class named ConfigParser. It contains an internal ConfigParser::Lexer class, which is responsible for detecting words and sections inside quotes, skipping comments, and keeping track of the position in the configuration file (line number and character number on that line for printing errors should there be any).

Dedicated functions are used for parsing logical parts (e.g., parseOption for parsing options) and these call each other (e.g., parseOption will call parseMatch if the option has a match as its child).

4.7.1 Other files and tree references

The public method `ConfigParser::parse(bool mainConfig = true)` is also used for parsing external files (a new `ConfigParser` instance gets created for that file). If the function encounters a library tree from a file, which has not been parsed yet, this file is parsed before continuing. To enable referencing trees in their file before they are defined, unknown tree references are not immediately reported as errors. Instead, they are kept by name in a static member `ConfigParser::_treesToFill`. It is a map with string keys (names of the trees) and values of a `struct ConfigParser::treePlacement`. This `struct` contains a vector with places to fill the tree and a vector of file positions (for error reporting when a tree is not found).

A place to fill is represented by a type named `treeToFillT`. It is a pair of a vector of `actions` and an index into that vector, where the tree should be. We do not just use a pointer because pointers into a vector are not persistent and these vectors are inserted into.

When parsing of a tree gets finished, it is checked if the tree is in `_treesToFill` and if so, it gets filled and removed from `_treesToFill`. At the end of parsing of the main config file, `_treesToFill` should be empty. If it's not, there is a tree that was referenced but not found, and an error is reported (the file positions where it was referenced are stored in `treePlacement::filePositions`).

4.7.2 `_keyword` member

Besides a `Lexer` instance, parsing methods share a string member of `ContextParser` named `_keyword`, which stores the last word read from the configuration.

Some of the parsing methods expect the `_keyword` member to be filled when they get called, and they treat it as the first word of their input. This behavior was introduced because callers of some of these methods need to read the keyword to determine what method to call. Namely, the following methods expect `_keyword` to be filled for them:

- `parseElement` — parses either a tree or an `input:` or `socket:` specification
- `parseTree` — the keyword will usually be the tree name, which needs to have been read by `parseElement` to determine that it's dealing with a tree and not an `input:` specification
- `parseActionList` — the behavior in this case is just to make implementation of `parseTree` slightly simpler
- `parseOption` — the keyword needs to be the opening brace of that option, which must have been read by `parseMatch` to find out that there are more options to parse

Other methods ignore what `_keyword` contains when they are called. Instead, they just continue reading from the configuration file (using the `Lexer` instance).

4.7.3 Comments on some of the `ConfigParser` methods

This section contains the description of a few chosen `ConfigParser` methods.

- `parseBlock` — This method parses a block enclosed in curly braces containing key-value pairs. It receives as its argument a map, which for every valid key stores a `std::function` processing its occurrence. This function can, for example, read the value and just store it somewhere.
- `parseActionList` — Parses a block containing a list of actions (written just after each other with no delimiter). It is used when parsing a root or a match. It takes a `std::vector<ParseAction *>` reference and puts the actions in it. It uses a map called `_parseFunctions` in a way that is similar to `parseBlock`. All supported actions have a handler for them stored in `_parseFunctions`. This handler is typically a lambda function, which calls the appropriate `ConfigParser` method (e.g., `parseMatch` for matches) and stores its returned value in the appropriate location. This way, if a new action is to be introduced, the `ConfigParser` only needs a parser for the action to be added to `_parseFunctions`.
- `parseExternalFile` — Creates a new `ConfigParser` instance and calls its `parse` method.
- `findExternalTree` — Handles a `parse_by:` specification. It receives an argument called `placeToFill` specifying where the found tree should be placed. It takes the `parse_by:` directive's argument, and it tries to find a tree with that identification. If there is no such tree, it splits the argument into file name (which may not be specified, in which case it's left empty) and the tree name itself. If a file name was specified and parsing of that file hasn't started, `parseExternalFile` is called. If a file was not specified, or its parsing has already started, `placeToFill` is added to places. These places are filled once the corresponding tree is parsed. This is why the method doesn't return the tree, and instead, it gets the `placeToFill` argument.

4.7.4 Default module and importing patterns

The way default modules and imported pattern files are handled stems from their scope, which is a subtree under the option where they were specified.

Each time a `default_module:` is encountered, a new element replaces its predecessor. When we finish parsing a subtree under an option, all of the `default_module:` directives specified in it (which are the k newest ones) lose their effect. This implies that default modules should be stored in a stack. It is implemented by a `std::vector ConfigParser::_defaultModuleNames`, which stores their names. When a `call:` or a `execute` directive is encountered, the last element in `_defaultModules` is passed to its constructor.

Removing elements from the stack is done inside `parseOption`. Any number of `default_module:` directives may be specified under a single option. To handle that, `parseOption` remembers the original size of `_defaultModuleNames`, and at its end, it pops all the elements it added.

We keep track of patterns to import in a way analogous to the default modules. In this case, however, the whole vector is passed to the appropriate constructor, because all of them are to be imported. The pattern files are then passed to `grok_patterns_import_from_file`. Since this function imports patterns for one specific `struct grok`, it is called for every `GrokIntegration` instance constructed. However, this is done before runtime, and the pattern files themselves have no dependencies that would have to be read again, so it should not be significant for the performance.

4.8 Timestamp parsing

`class TimestampParser`, is responsible for parsing timestamps. It inherits from `ParseAction` and is placed in the parsing tree in the place where the user wrote the `timestamp:` block. The parsing itself is done in its `perform` method.

The `strptime` function of the standard C library is used for parsing the text representation (without timezone and micros) into `struct tm`. The struct is cleared using `memset` in the beginning of `perform` to behave better if the format string omits something - e.g., does not contain a year field descriptor. `mktime` is then used to obtain the Unix timestamp from this struct. To handle time zones ourselves, we set the `TZ` environment variable to `UTC+0`, making `mktime` assume the time zone to be UTC. We then add the offset (if the user supplied it) manually to the result of `mktime`.

However, using `mktime` every time was rather slow, and therefore the last Unix timestamp is cached along with the last `struct tm`, and if the new `struct tm` has the same hour (and year and day), the timestamp is calculated from the previous timestamp. Since `mktime` assumes a constant time zone, this caching is correct and could even be used for instances of `struct tm` differing in their hour. There are plans to use the cache every time the instances have the same day (and year).

4.9 `break` implementation

The `break` keyword after an option means that other options under the same match should be skipped if the option is successfully matched. When walking through the tree, this breaking will be done inside the `perform` method of that option's father. A `break` after a match means that if any of its options are matched successfully, all actions following the match in the same block should be skipped. Since instances of `MatchOption` are only passive objects, this breaking will be done inside the `perform` method either of its father (if it is a root) or the match two levels above the match with the `break`.

The presence of the `break` keyword is signified by setting a bool flag (either in a `MatchOption`, or a `Match`). When `Match::perform` finds that an option was successfully matched, it checks if its flag is set. If yes, the rest of the options are skipped.

The `break` after a match is a little more complicated because the signature `void(Context &)` does not allow `Match::perform` to report whether it succeeded with at least one option. To get around this, we introduced a static flag inside

`Match`, which is set exactly if if the match is followed by `break` and was successful. This flag is set when returning from `Match::perform` and is immediately read and cleared by the `Match::perform` or `Root::perform` above.

5. Problems encountered

5.1 Speed considerations

We used a profiler named perf [1] to look for bottlenecks in the program. Since some of these bottlenecks seemed to be unnecessary syscalls, we also used the strace tool to get a list of syscalls made.

5.1.1 Grok

Grok is used for regular pattern matching and it uses a library named Tokyo cabinet [10]. This library offers, among other features, tree structures. Grok uses those to store parts of the expression by name. These trees are walked through in `grok_match_walk_next`, which we initially used for storing attributes when a match is successful. However, inside the Tokyo cabinet walk implementation, a custom Tokyo cabinet's assert is called, and this, in turn, calls `sched_yield`. This call overshadowed other syscalls in strace output and slowed down the run time.

Therefore we looked for a way to avoid calling `grok_match_walk_next` during runtime. It turned out that grok offers functions to walk over what it calls `grok_captures`. Each `grok_capture` contains a representation of that part of the input string matching a particular part of the regular expression — a pointer to a string (`char **`) pointing to an array and two indexes into this array — one representing the beginning of the string and the other for the end. The string can therefore be found out by simply indexing into that array.

To take advantage of this, a walk across `grok_captures` is performed in `GrokIntegration` constructor, and the found `grok_captures` are stored in a vector inside the `GrokIntegration` instance. During runtime, when the pattern is successfully matched, this vector is iterated over, and all of the attributes are assigned. This iteration through the vector is still visible in the profiler output but the time was improved.

Some thought was given to the possibility of avoiding even this iteration and attribute assignment by computing more at the time of configuration parsing. The idea was that instead of indexing into an array and assigning the `char *` into the context, a `char **` would be computed, which would only have to be dereferenced when accessing the attribute (probably actually two pointers — one for the beginning, one for the end). However, the attribute may be assigned in multiple options (which means there is no `char **` that would reliably point to the contents of the attributes) and attributes can be assigned from Python, so this approach was abandoned without implementation attempts.

5.1.2 Timestamp parsing

The `mktime` standard C library function is used for computing Unix timestamp from `struct tm`. In the early stages of implementation, the `TZ` environment variable was not being set by the program, which led to repeated `stat`-s of `/etc/localtime` (with every line of the log), which slowed down the execution. When we stopped assuming all times were in the current timezone and instead implemented custom timezone support by setting `TZ` to `UTC+0` and manually offsetting the computed time by the supplied time zone, this problem ceased. However, there still seemed to be a significant impact of mktime on the execution time. To address this issue, we cache the last `struct tm` used in `mktime` and the corresponding Unix timestamp. When converting to Unix timestamp, this cached value is checked, and if it has the same year, month, day and hour as the cached time, the last Unix timestamp is used, and the difference of the past and the current time is added to it.

5.1.3 I/O

Debugging prints were introduced to give the user an idea of how the parsing of both the configuration file and the input log lines. The early implementation took a `std::string` and decided whether or not to print this debugging message. This, however, meant that the string arguments got created even when the debugging prints were turned off, and the memory allocation inside it slowed the processing significantly (about seven times). The implementation was therefore changed to a variadic template taking `const char *` and `const str *` arguments to avoid the requirement for memory allocation.

Very early on in the implementation, `std::getline` was used instead of custom line reading and `std::string` instead of our `str`. This approach was very slow (probably due to the needed memory allocation but possibly partly due to the copying done after each successful match) and was quickly changed.

5.2 Reading from directory

It is important, especially for session tracking, to read the input files (should there be more of them) in the correct order, which needs to be specified by the user. Supporting arbitrary order (even based only on the file names) would need to involve Python, and in the end, it was deemed unnecessary. One idea was to order the files by their modification date. However, when files are rotated (or a new file gets created), it may happen that some messages are logged to the new file while other messages are written in the old one. Therefore the modification time is not a reliable indicator of the correct file order. There are two expected use cases: log rotation and log files with names based on the time of their creation. In these cases, numeric sorting by a number suffix (for rotation) and lexical order (for timestamped names) are sufficient, and therefore are the supported orders.

The fact that messages may be logged into two files simultaneously could also be problematic for reading messages since they might not be read in the correct order even if the files themselves are. However, each session will typically be linked to only a single process, and therefore messages associated with that session will be processed in chronological order. Therefore we believe that this will not be a major problem.

We support reading from multiple files (filtered by a name regexp) and saving progress for later resuming where we left off. The problem is that, in principle, any of the input files can grow (at any time — our program may or may not be running) and be renamed. Let us first deal with a single run without resuming saved progress.

Since file names are not reliable identifiers in our environment, we picked inodes instead — when a file is renamed, its inode stays the same (and if it's opened for reading, it stays open as if no renaming took place). When encountering the end of input of the last file, we may wish to terminate or continue monitoring the input files for growth or wait for new files to be created. In that case, we saw two approaches to take - either register to the system what updates we wish to be notified about using something like inotify, or look for new files (or added content) every once in a while and handle changes found. We implemented the latter approach because it seemed simpler to implement, but we think it might be nicer to register with the system for changes.

Since we don't want to make assumptions about which file is going to grow (or assume that new files are added only when other files do not grow), we check for new content or new files every time we reach the end of the last file. This means that if the last file is growing too fast to process and other files are growing too, then the content of those files may not be processed until the last file stops growing. However if this state is only temporary then this falls within the irregularities around switching log files. And if this state lasts longer, then it is not completely clear in which order the messages should be processed, as there are multiple independent processes logging simultaneously. We, therefore, think this behavior is reasonable.

Let us turn our attention to saving and resuming progress. As mentioned earlier, files are identified by inodes. We need new files to be read from the beginning and old files' inodes to be forgotten to prevent cluttering of the progress storage. To achieve this, when the program is terminating, it stores pairs of inode — file position into the progress file. When the program starts again, and a progress file is specified, it finds the list of files whose names match the requested pattern. Only these files are read and remembered. It looks for any progress stored for each of these files' inodes — if there is none, zero is assumed. The files are then opened in order, seek is performed to their position, and they are read to their end before continuing to the next file. When the program is terminating again, it only writes progress for those files which were found and whose name matched the pattern. Other files are forgotten, as required.

5.3 Python interface

Python gets called from the main program but it needs to use some functionality offered by the C++ implementation (like attribute access or calling library trees). So there need to be some functions that are callable from Python and can handle the appropriate C++ objects (like the context). The original approach was to compile a Python module (written in C++) using a Python build script and import that module. This posed a couple of problems.

The output of the compilation was a shared object file, which needed to be found at runtime. For development purposes, this could be done by setting the `LD_LIBRARY_PATH` environment variable to the directory containing the shared object file. This wasn't fit for production, where the project should be installed in a system directory and used as a command. Forcing the user to specify

LD_LIBRARY_PATH would hardly be acceptable, so the .so file would have to be placed in a different system directory to be found by the default search algorithm of the dynamic linker.

It also meant that some parts of the project needed to be linked to the Python module, for example, the context, which needs to be accessed by user scripts. The compilation flags used by the default Python compilation machinery made this a problem and the workaround we found was to compile these parts into a shared object file as well. This would mean another .so to take care of during installation.

We later changed our implementation to construct the Python module at run time using the Python C API.

Beaver manually buffers its output in an attempt to increase speed. This creates difficulties with Python's default print function, which does not write into our buffer. That means that messages from Python may drift from related messages printed through our buffering (especially as Python also buffers internally). This is hardly desirable since messages from the main program and from a Python script linked to a single common line of the log may appear far from each other and it might therefore seem that they are associated with different log lines. There needs to be a way to allow Python scripts to print via our buffer. This is done by adding a custom print function. However, the default print interface from Python offers much of functionality, like variable number of arguments, custom separation, and custom end character (the end argument, usually newline). We did not want to implement all this functionality ourselves as this would be time-consuming and would add little to our core functionality. We decided to only implement print and println functions. The former takes exactly one argument and converts it to string before printing. The latter takes one optional argument and processes it in the same way. The rest of the desired functionality can be achieved through Python's f-strings.

5.4 Miscellaneous

Grok allows the usage of predefined patterns using a syntax like %{regexp_name:attr_name}. However, when obtaining expression parts through grok_capture_walk_next, these parts of the pattern have names containing the colon and the regexp name (so the name in our example here would be regexp_name:attr_name). This is undesirable for our attribute assignment semantics, so the names are manually changed to only contain the name of the attribute.

When parsing a configuration file containing inline Python code, we need to deal with indentation as Python is whitespace-sensitive, and we expect configuration files to have their own indentation for better readability. We could force the user to scrap the config file indentation and write the Python code with proper absolute indentation. The result would, however, look something like this:

```
match: msg {
    { "hello world"
        execute {
ci.print("hello world encountered")
```

```
        }
    }
}
```

This dramatically reduces readability. Therefore, we use relative indentation: we edit the read code by removing some of its leading whitespaces. We could take the indentation to be relative to the position of the opening brace, but this would result in a rather large indentation. What's more, this indentation would be determined by the rather arbitrary number of characters in our keyword `execute` and how many spaces the user writes between the keyword and the opening brace. This could yield indentation by a number of spaces not divisible by four and would also complicate matters if tabs are used for indentation instead of spaces. We, therefore, take the base indentation from the first code line following the opening brace. This closing brace cannot be part of any valid Python code.

We also need to be able to detect the end of the Python code. This seems simple — the block is terminated by a closing curly brace. However, that character could be part of the Python code (inside a string literal). Therefore, we need to react to only those braces that cannot be part of a valid Python code. Thanks to relative indentation, we can simply wait for a closing brace that is indented by fewer spaces than the first code line.

6. Possible future improvements

Intermediate attributes

Currently, the attributes are categorized as intermediate or final ones, based on whether their name begins with an underscore. The idea behind this is to have internal implementation details separated from final outputs. We support context print, and in the future, we might support printing it in a structured way. In that case, however, we will hardly wish to print every single attribute that gets used. Some attributes should stay hidden.

However, this idea could be taken a step further. Intermediate attributes could be scoped within the tree where they are assigned. This would prevent name collisions among attributes from different trees, some of which can be library trees written by someone else. The scoping could be done by name mangling. We could include the name of a tree in the name of any private attribute, followed by the attribute's short name. To prevent name collisions with user-defined attributes, we could include a character the user cannot put in an attribute name. When parsing the configuration, we would have to remember the name of the tree we're currently in and adjust the attribute names accordingly. The intermediate attributes should probably be renamed to private attributes to reflect this semantic.

Template trees

To call a library tree now, we need to set the attributes the tree will use. Which attributes get used and for what is determined by the library tree itself and it is constant for one tree — the caller cannot do anything about it. The same holds for the results of the parsing, they get assigned to some fixed attributes which the caller can't choose.

This could potentially be improved by introducing some sort of template trees. A library tree could specify its template arguments, which would be names of attributes. The callers would set these template arguments in the place where they call the library tree. Another instance of the tree would then be created, which would use the appropriate attributes. This could, however, pose problems for using Python, so we might have to implement some way of accessing template argument values from Python.

Some thought should also be given to the question if any other types of template arguments should be supported. We could implement template arguments being other trees, which get used.

C modules

Python was chosen as our language for user programming to increase user comfort. However, in some cases, the run time could be critical, and the time spent in Python is significant. Users in such a situation might benefit from the addition of C/C++ modules. The user could supply a dynamic library, which would be linked dynamically and appropriate functions inside it called.

The interface would probably be similar to that used by Python scripts except that the strings would used would probably by instances `struct str` rather than `std::string` or null-terminated `char *`. The trick used in matching to get null-terminated strings without copying is to simply replace the character after the end of the matched string by `\0` and restore the original state after the matching is done. However, in the case of user modules, it could not be used since multiple attributes may be accessed by the user function. If we tried to put `\0` at the end of each, we would likely end up with some attributes containing `\0` inside them.

Explicit importing and tree name scoping

The current implementation imports library tree files when a tree from that file is used in a `parse_by` action. When the user wants to call a tree from Python, they must ensure that the tree is imported. In some cases this would mean creating another tree in the main config file, which would never get entered but it would contain the appropriate `parse_by`. It would be nicer to allow the user to import library tree files explicitly as well. We could even have a function for the user to call from Python which would ensure parsing of a specified configuration file. The user could call this function from a module at the time the module is imported (as described among the tips in user documentation), which happens at the same time other configuration files are parsed.

Library configuration files

We implemented a framework for parsing logs with support for a kind of libraries. However, we wrote no libraries for the user. For example, a tree for top-level parsing of messages conforming to one of the syslog RFCs could be a nice library tree to be supplied with the program. Additionally, we may supply a library tree for parsing some of the most common formats of timestamps and trees for some of the parsing of the logs we gave as an example at the beginning of chapter 2.

I/O improvements

It would be no doubt a useful feature to be able to print the output in some structured way, like JSON. It could be an action very similar to a context print except that it would print the attributes in JSON or some other format.

Furthermore, we could implement receiving syslog packets over UDP or TCP, not just through Unix socket, and therefore act as a full-blown syslog server.

Getting all attributes

There might seem to be a workaround for the structured output printing mentioned above — call a Python function, which will use some appropriate library to do the print. However, this function would have no way of getting the full list of available attributes. That is because attribute accessing is done through `__getattribute__`, which means that `dir` calls will not reveal any of them.

We should therefore support some way of accessing the list of attributes, probably by implementing `__dir__`.

Different attribute types

Some attributes don't have the semantics of a string. Initially, this only included the timestamp, so we decided to implement it as a one-time exception. However, the list expanded to include the facility and severity values when receiving messages through a socket using syslog. Therefore it might be better to implement support for attributes of different types.

These attributes could not be directly used for matching (without converting them into a string), so they would probably be mostly used by Python. Therefore for support of arbitrary attribute type, we might store attributes of type `PyObject *`. If these attributes were used in a match, we could either give a warning and consider all of the options to fail, or we could try to convert the attribute to a string using Python `__str__` function — that way, the user could implement custom conversion stringification.

The non-string attributes we have encountered so far, though accessed mostly by Python, were created by Beaver itself (timestamp by `TimestampParser` and priority and severity values were set in one of the functions responsible for reading input). They could, however, be converted to `PyObject *` just after creation.

We could also try to serialize values of these attributes to a sequences of bytes and convert them to `PyObject *` when they are accessed by Python and from `PyObject *` to the value when they are set by Python. In this case, we would have to be careful not to interfere with Python's garbage collection when non-string attributes get created by Python. We would have to keep ownership of all references it holds to make sure those objects aren't deleted. We would also need to keep track of any changes to the serialized object, and if some of the references it has started pointing to a different object, decrease the appropriate reference count (otherwise, we could cause a memory leak). This approach seems fragile, so maybe we should keep at least the attributes created by Python as simple `PyObject` pointers.

7. Performance and comparison to other tools

This chapter aims to compare Beaver's speed and set of features to other tools that could be used for a similar task.

The speed testing was done inside on a laptop inside a Podman container. The laptop was had an 8-core `Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz` with a cache size of `6144 KB`.

Two input files were used. The smaller one was a real Postfix log, and the larger one (which was used where possible) was the smaller input file concatenated 200 times with itself, making it roughly 1GB in size.

The script used for benchmark and the output it produced are in attachments. It ran each tool in each of the configurations 10 times and recorded the run times. The script then calculated the minimum, maximum, and average run time. This measurement was done twice, and the results of the first one were not kept. This was to let the kernel cache the inputs etc. The output of the script (see attachments) contains all 10 of the measured times for each tool and configuration.

7.1 Profiler output

Perf [1] was used for profiling with the larger input file. Its output for the counter example (the one counting the number of e-mails sent every minute) is the following:

```
perf report --sort=dso
  66.07%  libpcre.so.3.13.3
  15.33%  libpython3.8.so.1.0
  11.39%  beaver
   5.69%  libc-2.31.so
   0.94%  libgrok.so.1
   0.36%  ld-2.31.so
   0.23%  [unknown]
```

We can see that most of the time is spent inside PCRE, which does pattern matching. Another significant item is Python, which can be expected since Python is not as efficient as C++. Let us look at the other two noteworthy entries — Beaver itself and libc.

```
perf report --dso=beaver
Overhead  Command  Symbol
   2.59%  beaver   [.] parsing::Context::reinitialize
   1.70%  beaver   [.] parsing::GrokIntegration::tryMatch
   0.97%  beaver   [.] parsing::Context::getAttribute
   0.80%  beaver   [.] parsing::TimestampParser::perform
   0.48%  beaver   [.] parsing::GrokIntegration::tryMatch
   0.32%  beaver   [.] std::unique_ptr<std::map<parsing::str,
   0.32%  beaver   [.] std::vector<grok_capture const*,
   0.32%  beaver   [.] std::_Rb_tree<parsing::str,
   0.31%  beaver   [.] std::swap<std::__cxx11::list

perf report --dso=libc-2.31.so
   1.88%  beaver   [.] __strptime_internal
   1.12%  beaver   [.] __strlen_avx2
   0.81%  beaver   [.] __strcmp_avx2
   0.76%  beaver   [.] __memchr_avx2
   0.70%  beaver   [.] __GI_____strtoll_l_internal
   0.16%  beaver   [.] __GI_strtoll
   0.09%  beaver   [.] __wcslen_avx2
   0.08%  beaver   [.] __memmove_avx_unaligned_erms
   0.08%  beaver   [.] cfree@GLIBC_2.2.5
```

Most of the time in Beaver itself was spent inside the following three methods:

- `reinitialize` — Sets all attributes to empty strings after each message is parsed.
- `tryMatch` — Tries to match an attribute against a pattern. This method has to iterate through all the attributes which should be assigned.
- `getAttribute` — This one is called every time we need to access an attribute.

There are other items in the output, but they take up less than 0.3% of time each.

The first item in libc is `strptime`, which parses the timestamp, followed by `strlen`, `strcpm`, and `memchr`. It seems that `strlen` is used by PCRE during matching. `memchr` is used for detecting line ends and therefore goes through the whole buffer after every read. It is also the place where we can expect most of the cache misses to occur.

This was the output for the reorder example:

```
perf report --sort=dso
  39.67%  libpython3.8.so.1.0
  29.41%  libpcre.so.3.13.3
  14.81%  beaver
  12.64%  libc-2.31.so
   1.46%  libstdc++.so.6.0.28
   0.82%  ld-2.31.so
   0.72%  libgrok.so.1
   0.46%  [unknown]

perf report --dso=beaver
   2.45%  beaver   [.] parsing::TimestampParser::perform
   1.65%  beaver   [.] std::_Rb_tree<std::pair<_object*,
   1.42%  beaver   [.] parsing::print
   1.39%  beaver   [.] parsing::str::operator<
   1.23%  beaver   [.] parsing::python::getAttribute
   1.22%  beaver   [.] parsing::GrokIntegration::tryMatch
   0.99%  beaver   [.] parsing::Context::reinitialize
   0.72%  beaver   [.] parsing::Context::getAttribute
   0.72%  beaver   [.] std::_Rb_tree<std::pair<unsigned long long,
   0.64%  beaver   [.] parsing::Context::setAttribute
   0.48%  beaver   [.] parsing::Context::getAttribute
   0.25%  beaver   [.] parsing::DebugPrinter::isEnabled
   0.25%  beaver   [.] 0x0000000000005ff0
   0.25%  beaver   [.] 0x0000000000005b04
   0.25%  beaver   [.] readFrom
   0.25%  beaver   [.] 0x0000000000005ba4
   0.25%  beaver   [.] parsing::Root::perform
   0.24%  beaver   [.] parsing::python::endSession
   0.18%  beaver   [.] 0x0000000000006104
```

In this case, Beaver takes up more of the time than previously but still less than Python (which takes up a lot more time than in the previous case) and PCRE. We can see that compared to the previous case, a lot of the time was spent in `print`, which gets invoked a lot in here, but it is still Beavers function, not the actual syscall. This could be potential room for improvement. Also, `str::operator<` takes up a significant amount of time. It is most likely in the search for an attribute index since keys in the associated `std::map` (used by Python) are of type `str`. This may improve if we used `std::unordered_map` instead and implementing some reasonable hashing of `parsing::str`.

Keep in mind that these results differ among runs. This is another run of the counter example:

```
  71.24%  libpcre.so.3.13.3
  15.40%  libpython3.8.so.1.0
   7.30%  libc-2.31.so
   4.84%  beaver
   0.50%  libgrok.so.1
   0.46%  ld-2.31.so
   0.20%  [unknown]
   0.05%  libpthread-2.31.so
   0.02%  libm-2.31.so
```

And this one is an additional execution of the reorder example:

```
  44.86%  libpython3.8.so.1.0
  28.97%  libpcre.so.3.13.3
  14.38%  beaver
   9.81%  libc-2.31.so
   1.11%  libstdc++.so.6.0.28
   0.73%  ld-2.31.so
   0.13%  [unknown]
```

7.2 Grok

Grok [18] is primarily a tool for parsing logs into structured data. Its configuration allows its user to perform something similar to the part we have called "parsing" in this text. The difference is that with Grok, there seems to be only one match taking place. The results are then passed to one of the supplied filters, `jsonencode` being the most noteworthy one — it encodes a string into a JSON string (escaping special characters in it). The user can therefore use `jsonencode` together with the parsing to convert the log into a series of JSON objects, each one representing one log message. Grok also goes with a tool called GrokDiscovery, which seems to try to find known patterns in the input. The so-called Grok programs have inside them a command, which they run and read its output.

Beaver, on the other hand, allows analysis to be built-in and parsing to be done with the help of Python. Reading the output of another process can be done through a pipe or, in some cases, the Unix socket.

For speed comparison we used the following Grok config (line breaks in patterns are again purely typographical and aren't in the real config):

```
program {
  exec "cat input_large.in"

  match {
    pattern: "%{time=[[:digit:]-]*T[^+.]*}\.%{micros=[^+]*}\+
%{offset=[^ ]*} %{message=.*}"
    reaction: "message : %{message}\nmicros : %{micros}\noffset :
%{offset}\ntime : %{time}\n"
  }
}
```

and this Beaver config:

```
main {
match: msg {
    { "%{time=[[:digit:]-]*T[^+.]*}\.%{micros=[^+]*}\+%{offset=[^ ]*}
%{message=.*}"
        print: context
    }
}
}
```

The run times are the following:

tool	fastest time	slowest time	average time
Grok	64.03 s	71.91 s	67.83 s
Beaver	10.98 s	11.30 s	11.08 s

So Beaver is actually significantly faster for this configuration. Since Beaver and Grok use the same library for regular expression matching, this performance comparison will not be sensitive to the pattern used.

Let us note that the output of Beaver and Grok differs somewhat: there some lines present in the Beaver output, which are not in Grok output. These lines form a continuous suffix of the output and there are no other differences. However, there are some lines missing in the output of the Grok configuration, which just echoes its input, i.e.:

```
program {
  exec "cat input_large.in"

  match {
    pattern: ".*"
    reaction: "%{@LINE}"
  }
}
```

These lines also form a continuous suffix (even though the input was our larger one, which is periodic). Therefore, we tend to think this is a bug in Grok. It can be reproduced with `seq 10000` as the input. The corresponding Beaver configuration below reproduces the output exactly.

```
main {
match: msg {
    { ".*"
        print: "{msg}"
    }
}
}
```

7.3 Logstash

Logstash [6] can read from a large variety of input kinds, like file, syslog and Redis using its input plugins [8]. The log messages become so-called "events" and are passed through a series of filters [7], which may besides literal filtering add "fields" to the event. These fields seem to be similar to our attributes. After passing through the filters, events are sent to output plugins [9], which may write them to standard output, hand them over to another program, etc.

In particular, Grok is offered among the filters. Another filter that is interesting to us is the Ruby filter, which allows the user to either execute a Ruby script or an inline Ruby code. The user program gets a Logstash event and it returns an array of events. This is somewhat similar to the way Python is used inside Beaver. Beaver, however, calls different functions from the supplied script, whereas Logstash seems to call one particular function named `filter`. Also, the Beaver interface is that user code modifies a context and possibly calls a library tree instead of generating an array of contexts.

We haven't found an easy way to write something similar to Beaver's parsing tree in Logstash. It seems that there can only be a series of filters, all of which are applied. Events can be dropped and therefore hidden from a filter, but we found no way of picking them up in the following filters once they are dropped.

7.4 RSyslog

RSyslog [3] is primarily a syslog server. Like Logstash, it offers a number of plugins, which may filter and alter messages or deliver them to other processes, such as a database. The messages are processed by so-called "rules", each of which consists of a "filter" and a list of "actions" [5].

There is the possibility to execute an arbitrary program through its `omprog` module [4], but their connection seems to be looser than Beaver's `call` action — RSyslog provides command-line arguments to the program and then possibly reads what the program wrote on its standard output. It seems to be intended for reacting to log messages rather than helping to process them (this is supported by the fact that `omprog` is considered an output module).

Filters include, for example, matching against a regular expression and checking whether a string starts with a particular string. It is also possible to filter messages based on their properties, e.g., the hostname contained in the message. However, it seems that RSyslog detects these properties itself without letting the user configure this detection.

Therefore, Beaver's user programming support seems better for aiding message processing, and Beaver also supports tree structures. It seems to us that this is not the case with RSyslog. Additionally, there seems to be only limited support for processing the message as a structured entity.

7.5 sed, awk and grep

Sed and awk are traditional Unix tools for processing text. They are not specifically aimed at logs but could be useful in cases where only simple alteration or filtering is needed. Specifically, sed configuration would be unreadable if it was used for complex processing. Grep only filters lines of input, we included it here as a performance benchmark.

For comparison with grep we used the Beaver configuration below, which filters messages containing `removed`.

```
main {
match: msg {
    { "removed"
        print: "{msg}"
    }
}
}
```

We also tested sed and awk on the same task. We used the following configurations:

tool	**config**
grep	`removed`
sed	`/removed/p`
awk	`/removed/{ print $0}`

The run times are the following:

tool	**fastest time**	**slowest time**	**average time**
grep	0.88 s	0.89 s	0.88 s
sed	1.95 s	1.96 s	1.96 s
awk	1.15 s	1.16 s	1.15 s
Beaver	2.47 s	2.49 s	2.48 s

Beaver is about three times slower than grep, which is the fastest, about two times slower than awk and somewhat slower than sed.

Next, we tested configurations where the text is not only filtered but also modified. We used the following config to filter messages from Postfix/smtpd and print them without their header:

```
main {
match: msg {
    { "postfix\/smtpd\[[0-9]*]: %{msg=.*}"
        print: "{msg}"
    }
}
}
```

Beaver uses named captures in this config, and sed offers a feature, which is somewhat similar, albeit notoriously slow - back-references. So we tested sed with that feature. So we tested two sed configurations — one regular and one with back-references.

These are the configurations we used:

tool	**config**
sed	`s/.*postfix\/smtpd\[[0-9]*]:  //p`
awk	`{if(sub(".*postfix/smtpd\[[0-9]*]:  ","") > 0) print $0}`
sed (bref)	`s/.*postfix\/smtpd\[[0-9]*]:  \(.*\)/\1/p`

With the following results:

tool	**fastest time**	**slowest time**	**average time**
sed	6.46 s	6.65 s	6.51 s
awk	2.70 s	2.77 s	2.72 s
sed (bref)	60.12 s	60.31 s	60.22 s
Beaver	4.40 s	4.43 s	4.42 s

It seems that Beaver can be somewhat faster than sed under some circumstances, significantly faster if back-references are needed. Awk was somewhat faster than Beaver in our experiment.

However, please keep in mind that Beaver's performance may differ significantly among some patterns (as described in subsection 3.9.7). Since sed, awk, and grep are unlikely to use the same regexp library as Beaver, we may get different results if we use a different pattern.

7.6 Python

Python is a general-purpose programming language, but it does offer a regular expression library, which even contains named captures. However, the tree structure that Beaver supports would require some amount of work to implement and this work would probably have to be done again every time. Additionally, Beaver offers built-in session support with a timeout, reading from a directory or a socket. All of these could obviously be implemented in Python, but it would require some additional effort.

We took the example counting the number of e-mails every minute described in chapter 3. We chose the third variant of the analysis — the one with only a single registered session to write the number of e-mails in the last minute. We implemented this example in Python without any help from Beaver and compared it to the example. This was the python script:

```
import re
```

```
from datetime import datetime

root = re.compile("(?P<time>[0-9-]*T[^+.]*)\.(?P<micros>[^+]*)\+
(?P<offset>[^ ]*) (?P<message>.*)")
message_match = re.compile("[0-9A-Z]+: removed$")

counter = 0
current_minute = None

def inccounter(t):
    global counter
    update_time(t)
    counter += 1

def print_counter():
    global counter
    global current_minute
    print(f"{current_minute} {counter}")
    counter = 0
    current_minute += 60

def update_time(t):
    global current_minute
    t = int(t.timestamp())
    t -= t%60
    if current_minute is None: current_minute = t
    while current_minute < t: print_counter()

def processline(line):
    global counter
    mr = re.match(root,line)
    if mr is None: return
    mr = mr.groupdict()
    timestamp = mr["time"]
    time = datetime.strptime(timestamp,"%Y-%m-%dT%H:%M:%S")
    mr2 = re.search(message_match,mr["message"])
    if mr2 is None:
        update_time(time)
    else:
        inccounter(time)

with open("input.in","r") as infile:
    for line in infile.readlines():
        processline(line)
print_counter()
```

The script uses the same regular expressions as the Beaver configuration, parses the timestamp and counts the number of e-mails in each minute just like the Beaver configuration.

The following is the speed comparison between the Python script and the original Beaver configuration. It was done on an input smaller than the previous measurement since that file was an original log concatenated multiple times with itself, and therefore its timestamps weren't in chronological order.

tool	**fastest time**	**slowest time**	**average time**
Python	0.60 s	0.63 s	0.61 s
Beaver	0.14 s	0.14 s	0.14 s

We can see that Beaver is significantly faster.

For comparison, we also ran the reorder example, which was introduced in chapter 2 and the slowest of the previous tests — sed with back-references. The exact configuration of the counter example is attached to this thesis, together with the source files.

The results were the following:

tool	**fastest time**	**slowest time**	**average time**
sed (bref)	0.30 s	0.30 s	0.30 s
Beaver	0.10 s	0.10 s	0.10 s

Conclusion

Beaver is a highly configurable tool for processing logs. The approach of handling syntax parsing separately from semantic analysis, coupled with the use of a full-fledged programming language for the analysis, gives the tool a considerable degree of flexibility. Its support for re-using parts of the configuration as libraries improves the flexibility even further.

Besides reading from standard input, the tool can receive log messages through a Unix socket or read them from files.

Beaver significantly outperforms Grok, an existing log parsing tool, when both perform the same task. The performance with a simple configuration even seems to be similar to tools like sed.

Compared to other log handling tools (like RSyslog or Logstash), Beaver lacks direct support for some inputs and outputs (e.g., reading from a database or writing to it), but it seems to offer more functionality for semantic analysis.

The tool could be further improved by adding support for receiving syslog messages over the network, printing output in a structured format (like JSON), or perhaps C modules, which could be used instead of Python ones when higher performance is required.

Bibliography

[1] Perf Wiki. [online]. Available at: `https://perf.wiki.kernel.org/index.php/Main_Page`. [Accessed: 2021-05-10].

[2] Regular expressions library. [online], 2020. Available at: `https://en.cppreference.com/w/cpp/regex`. [Accessed: 2021-05-19].

[3] Adiscon GmbH. RSyslog Documentation. [online], 2008. Available at: `https://www.rsyslog.com/doc/v8-stable/index.html`. [Accessed: 2021-05-10].

[4] Adiscon GmbH. RSyslog Documentation: omprog: Program integration Output module. [online], 2008. Available at: `https://www.rsyslog.com/doc/v8-stable/configuration/modules/omprog.html`. [Accessed: 2021-05-10].

[5] Adiscon GmbH. RSyslog Documentation: Basic Structure. [online], 2008. Available at: `https://www.rsyslog.com/doc/v8-stable/configuration/basic_structure.html`. [Accessed: 2021-05-10].

[6] Elasticsearch. Logstash. [online], 2021. Available at: `https://www.elastic.co/logstash`. [Accessed: 2021-05-10].

[7] Elasticsearch. Logstash: Filter plugins. [online], 2021. Available at: `https://www.elastic.co/guide/en/logstash/current/filter-plugins.html`. [Accessed: 2021-05-10].

[8] Elasticsearch. Logstash: Input plugins. [online], 2021. Available at: `https://www.elastic.co/guide/en/logstash/current/input-plugins.html`. [Accessed: 2021-05-10].

[9] Elasticsearch. Logstash: Output plugins. [online], 2021. Available at: `https://www.elastic.co/guide/en/logstash/current/output-plugins.html`. [Accessed: 2021-05-10].

[10] FAL Labs. Tokyo Cabinet: a modern implementation of DBM. [online], 2006. Available at: `https://dbmx.net/tokyocabinet/`. [Accessed: 2021-05-19].

[11] Gerhards, R. The Syslog Protocol. [online], 2009. RFC 5424. Available at: `https://rfc-editor.org/rfc/rfc5424.txt`. [Accessed 2021-05-10].

[12] International Organization for Standardization. Data elements and interchange formats — Information exchange — Representation of dates and times. [not publicly available], 1988. ISO 8601.

[13] John Maddock. Boost.regex 5.1.3. [online], 1998. Available at: `https://www.boost.org/doc/libs/1_66_0/libs/regex/doc/html/index.html`. [Accessed: 2021-05-19].

[14] Lonvick, C. M. The BSD Syslog Protocol. [online], 2001. RFC 3164. Available at: `https://rfc-editor.org/rfc/rfc3164.txt`. [Accessed 2021-05-10].

[15] Newman, C. and Klyne, G. Date and Time on the Internet: Timestamps. [online], 2002. RFC 3339. Available at: `https://rfc-editor.org/rfc/rfc3339.txt`. [Accessed 2021-05-25].

[16] Philip Hazel. PCRE — Perl Compatible Regular Expressions. [online], 2015. Available at: `http://www.pcre.org/`. [Accessed: 2021-05-19].

[17] Python Software Foundation. re — Regular expression operations. [online], 2001. Available at: `https://docs.python.org/3.8/library/re.html`. [Accessed: 2021-05-25].

[18] Sissel, J. grok. [online]. Available at: `https://code.google.com/archive/p/semicomplete/wikis/Grok.wiki`. [Accessed: 2021-05-10].

[19] Yergeau, F. UTF-8, a transformation format of ISO 10646. [online], 2003. RFC 3629. Available at: `https://rfc-editor.org/rfc/rfc3629.txt`. [Accessed 2021-05-25].

A. Attachments

A.1 Benchmark outputs

This is the output of the benchmarking script. The script itself is also attached but only in its electronic form.

```
/usr/bin/time beaver greplike.conf
    <input_large.in 2>&1 >output.ignore
min max avg: 2.47 2.49 2.48
times:  2.48 2.48 2.48 2.48 2.49 2.47 2.49 2.48 2.48 2.48

/usr/bin/time grep removed
    <input_large.in 2>&1 >output.ignore
min max avg: 0.88 0.89 0.88
times:  0.89 0.89 0.89 0.88 0.88 0.89 0.88 0.88 0.88 0.88

/usr/bin/time sed -n /removed/p
    <input_large.in 2>&1 >output.ignore
min max avg: 1.95 1.96 1.96
times:  1.95 1.96 1.95 1.95 1.95 1.96 1.96 1.96 1.96 1.95

/usr/bin/time awk '/removed/{print $0}'
    <input_large.in 2>&1 >output.ignore
min max avg: 1.15 1.16 1.15
times:  1.16 1.15 1.16 1.15 1.16 1.15 1.15 1.15 1.15 1.15

/usr/bin/time beaver sedlike.conf
    <input_large.in 2>&1 >output.ignore
min max avg: 4.40 4.43 4.42
times:  4.43 4.40 4.42 4.40 4.42 4.43 4.42 4.41 4.42 4.40

/usr/bin/time sed -n 's/.*postfix\/smtpd\[[0-9]*]: //p'
    <input_large.in 2>&1 >output.ignore
min max avg: 6.46 6.65 6.51
times:  6.47 6.50 6.46 6.47 6.49 6.65 6.50 6.46 6.50 6.62

/usr/bin/time sed -n 's/.*postfix\/smtpd\[[0-9]*]: \(.*\)/\1/p'
    <input_large.in 2>&1 >output.ignore
min max avg: 60.12 60.31 60.22
times:  60.17 60.22 60.25 60.28 60.21 60.15 60.31 60.29 60.12 60.20

/usr/bin/time awk
    '{if(sub(".*postfix/smtpd\[[0-9]*]: ","") > 0) print $0}'
    <input_large.in 2>&1 >output.ignore
min max avg: 2.70 2.77 2.72
times:  2.71 2.76 2.77 2.70 2.71 2.71 2.70 2.70 2.72 2.71

/usr/bin/time beaver groklike.conf
    <input_large.in 2>&1 >output.ignore
min max avg: 10.98 11.30 11.08
```

```
times:  11.08 11.04 11.30 11.02 11.14 11.18 11.04 10.99 10.98 11.00

/usr/bin/time grok -f beaverlike.grok
    <input_large.in 2>&1 >output.ignore
min max avg: 64.03 71.91 67.83
times:  64.03 65.94 66.74 65.23 66.36 71.91 69.64 69.39 69.50 69.59

/usr/bin/time sed -n 's/.*postfix\/smtpd\[[0-9]*]: \(.*\)/\1/p'
    <input.in 2>&1 >output.ignore
min max avg: 0.30 0.30 0.30
times:  0.30 0.30 0.30 0.30 0.30 0.30 0.30 0.30 0.30 0.30

/usr/bin/time beaver examples/counter.conf
    <input.in 2>&1 >output.ignore
min max avg: 0.14 0.14 0.14
times:  0.14 0.14 0.14 0.14 0.14 0.14 0.14 0.14 0.14 0.14

/usr/bin/time python3 counter_python.py
    <input.in 2>&1 >output.ignore
min max avg: 0.60 0.63 0.61
times:  0.61 0.60 0.62 0.61 0.61 0.60 0.60 0.63 0.63 0.60

/usr/bin/time beaver examples/reorder.conf
    <input.in 2>&1 >output.ignore
min max avg: 0.10 0.10 0.10
times:  0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10
```

A.2 Source code

A repository with the source code is attached to this thesis in the electronic form only. Some example configuration files and an example input are included.