4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

4.1 Алгоритм работы функции process_received_events монитора

Функция `process_received_events` предназначена для обработки и отправки событий файловой системы в отслеживаемых директориях. Она принимает в качестве аргументов объект контекста Inotify – `monitor`, менеджер обмена сообщениями - `monitor_messaging_manager`, список, предназначенный для возможных парных событий - `pair_events_list`, и объект события `Event` - `stop_events_processing_flag`. Сама функция выглядит следующим образом:

```
def process_received_events(monitor, \
        messaging_manager, pair_events_list, \
        stop_events_processing_flag):

    for event in monitor.event_gen():
        if event is not None:
            messages = process_possible_pair_event(event, \
                pair_events_list)

            if messages is None:
                message = get_message_by_event(event)
                messaging_manager.send_message(str(message))
            else:
                for message in messages:
                    messaging_manager.send_message( \
                       str(message))
        else:
            if stop_events_processing_flag.is_set():
                stop_events_processing_flag.clear()
                break
```

Основная работа функции происходит в цикле считывания событий из объекта контекста Inotify:

```
for event in monitor.event_gen():
```

Метод `event_gen` возвращает объект события если оно есть в очереди событий и None если очередь пуста. После цикла идёт проверка того – появилось новое событие или нет. Если оно появилось – мы передаём событие и список возможных парных событий функции `process_possible_pair_event`. Она возвращает пустой список, если текущее событие первое из парных, один объект

сообщения если в списке уже было парное событие, а текущее событие также парное, два объекта сообщения, если текущее событие не парное, а в списке уже было возможно парное событие и `None`, если список пуст и текущее событие не парное.

Далее проверяется вернула ли функция обработки возможных парных событий какие-либо сообщения:

```
if messages is None:
```

Если сообщения\сообщений нет – объект сообщения получается по событию с помощью функции `get_message_by_event` и отправляется в виде строки с помощью метода `send_message` объекта менеджера пересылки сообщений:

```
message = get_message_by_event(event)
messaging_manager.send_message(str(message))
```

Если сообщения есть – они также отправляются через объект менеджера пересылки сообщений, но уже в цикле:

```
for message in messages:
    messaging_manager.send_message(str(message))
```

В ситуации, когда событие не пришло и метод `event_gen` возвратил None проверяется флаг выхода из функции - `stop_events_processing_flag`, который устанавливается таймером раз в 10 секунд извне.

```
stop_event_timer = Timer(10, stop_event.set)
stop_event_timer.start()
```

Этот флаг нужен для того что бы обработка событий не занимала всё время монитора и каждые 10 секунд монитор имел бы возможность проверять входящие сообщения. Такое неравное распределение времени сделано потому что входящие сообщения присылаются гораздо реже чем отсылаются исходящие.

4.2 Алгоритм работы функции process_possible_pair_event монитора

Функция process_possible_pair_event предназначена для обработки событий, которые могут быть парными и для которых нужно генерировать одно сообщение вместо двух. Примером такого события может служить переименование файла, которое состоит из пары событий: файл перемещён из

директории и файл перемещён в директорию. Эту ситуацию надо отличать от событий реального перемещения файла. Код данной функции представлен ниже:

```
def process_possible_pair_event( \
        current_event, pair_event_list):
    (header, _, watch_path, _) = current_event

    if len(pair_event_list) == 0 and is_first_pair_event( \
            header):
        pair_event_list.append(current_event)
        return []
    elif len(pair_event_list) != 0:
        previous_event = pair_event_list.pop()
        (_, _, previous_watch_path, _) = previous_event

        if previous_watch_path == watch_path and \
                is_second_pair_event(header):
            pair_events_message = get_message_by_event_pair(
                previous_event, current_event)
            return [ pair_events_message ]
        elif is_first_pair_event(header):
            pair_event_list.append(current_event)
            previous_event_message = get_message_by_event( \
                previous_event)
            return [ previous_event_message ]
        else:
            previous_event_message = get_message_by_event( \
                previous_event)
            current_event_message = get_message_by_event( \
                current_event)
            return [ previous_event_message, \
                current_event_message ]
    return None
```

Функция принимает в качестве аргументов текущее событие `current_event` и список с возможными парными событиями - `pair_event_list`.

Объект `current_event` представляет собой кортеж, из которого нужно выделить заголовок с метаданными события и путь к отслеживаемой директории.

```
(header, _, watch_path, _) = current_event
```

Далее проверяется список возможных парных событий. Если список пуст, проверяется может ли текущее событие являться первым из парных. Проверку осуществляет функция `is_first_pair_event`, в которую передаётся

заголовок с метаданными о событии. Если оба условия соблюдаются – событие добавляется в список `pair_event_list` и функция возвращает пустой список.

```
pair_event_list.append(current_event)
return []
```

Если в списке уже есть первое парное событие, то оно извлекается и раскладывается на составные части для получения пути к директории:

```
previous_event = pair_event_list.pop()
(_, _, previous_watch_path, _) = previous_event
```

Возникает три ответвления для трёх разных ситуаций. Для первой ситуации проверяется совпадает ли имя отслеживаемой директории и, с помощью функции `is_second_pair_event`, является ли оно вторым парным. В случае если оба условия соблюдены – с помощью функции `get_message_by_event_pair` получаем один объект сообщения для двух событий и возвращаем его в виде списка.

```
pair_events_message = get_message_by_event_pair(\
    previous_event, current_event)
return [ pair_events_message ]
```

Для второй ситуации с помощью функции `is_first_pair_event` проверяется является ли текущее событие первым из парных. Если условие соблюдается – текущее событие добавляется в список, а предыдущее событие конвертируется в объект сообщения с помощью функции `get_message_by_event` и возвращается в виде списка.

Если первые две ветки условного оператора не сработали – значит текущее и предыдущее события непарные. В таком случае они оба конвертируются в отдельные сообщения, которые возвращаются в виде списка:

```
previous_event_message =get_message_by_event(previous_event)
current_event_message = get_message_by_event(current_event)
return [ previous_event_message, current_event_message ]
```

Если список пуст и текущее событие не является первым парным – возвращается значение `None`, что даёт понять вызывающему коду что текущее событие нужно обрабатывать обычным образом.

В итоге, результатом выполнения функции при таком алгоритме является либо список, который может быть пустым или содержать объекты сообщений либо значение `None`.

4.3 Алгоритм работы метода run потока приёма и передачи сообщений

Метод run класса `MessagingManagerBackgroudReceiver`, представляющего собой поток `Thread`, предназначен для последовательного приёма и передачи сообщений. Этот метод запускается при старте потока и при его завершении поток завершает свою работу. Код метода выглядит следующим образом:

```
def run(self):
    polling_delay = 100

    while not self._stop_event.is_set():
        if self._send_queue.qsize() != 0:
            self._socket.send_unicode( \
                self._send_queue.get())
        events = self._socket.poll(polling_delay)
        if events != 0:
            self._receive_queue.put( \
                self._socket.recv(flags=zmq.NOBLOCK))
    self._socket.close()
    self._context.destroy()
```

При старте этого метода запускается бесконечный цикл, который работает до тех пор, пока внешний код, создавший данный поток не установит событие, которое позволит выйти из цикла и корректно освободить ресурсы.

На первом этапе цикла проверяется есть ли в очереди `self._send_queue` сообщения на отправку. Если очередь не пуста одно сообщение из очереди отправляется с помощью метода `send_unicode` сокета `self._socket`. Если очередь пуста – цикл переходит к следующему этапу.

```
if self._send_queue.qsize() != 0
    self._socket.send_unicode( \
        self._send_queue.get())
```

На втором этапе идёт попытка получить сигнал о пришедшем сообщении за промежуток времени 100 мс. Такую возможность предоставляет метод сокета `poll`, куда в качестве задержки передаётся константа `polling_delay` и который возвращает 0, если за заданное время никаких сообщений не пришло. Если сообщение пришло – оно принимается методом сокета `recv` и помещается в очередь принятых сообщений `self._receive_queue`.

После того как вызывающий код выставил событие `_stop_event` и цикл завершился, освобождаются все ресурсы потока, а именно: контекст и сокет ZeroMQ.

4.4 Алгоритм работы метода get_rules менеджера правил