ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте на тему:

Оповещения через мессенджер Telegram для сервера мониторинга collectd

Выполнил студент:

группы #БПМИ2110, 3 курса Аникиев Ян Вячеславович

Принял руководитель проекта:

Корнилов Матвей Викторович
Кандидат физико-математических наук
Доцент факультета физики НИУ ВШЭ

Москва 2024

Содержание

Аннотация

В данной курсовой работе реализована библиотека на языке C, которая интегрируется в сервер мониторинга collectd и перенаправляет системные уведомления через API Телеграма в выбранные чаты. Также получившийся плагин позволяет читать входящие сообщения в Телеграм и отправлять в ответ готовую конфигурацию для collectd, используя long polling или webhooks. Итогом работы является Pull request в официальный репозиторий проекта collectd.

Ключевые слова

Оповещения, боты, телеграм, сервер мониторинга, язык C, многопоточность, динамическая библиотека, парсинг, прокси.

1 Введение

Проект collectd [3] – это легковесный open-source сервер мониторинга для операционной системы Linux, написанный на языке C и имеющий модульную структуру. То есть отдельные возможности реализуются плагинами в виде динамически загружаемых shared-object модулей. Collectd получает метрики (например, загрузка CPU или количество используемой оперативной памяти) и записывает их в ту или иную базу данных, или передает другим сервисам с использованием сетевых протоколов. Пользователь может настроить набор триггеров, приводящих к генерации оповещений (которые, в свою очередь, могут быть записаны в системный журнал, отправлены по электронной почте и т.п.). Пример использования оповещений: свободное дисковое пространство опустилось ниже определённого уровня, нужно привлечь внимание системного администратора.

Телеграм – это крайне популярный сегодня мессенджер. В Телеграме есть возможность создать своего бота и отправлять сообщения от его имени, используя Telegram Bot API [9]. Можно предположить, что многим системным администраторам было бы удобно получать нотификации о проблемах в сети именно через сообщения от телеграм-бота. Таким образом, возникает потребность в реализации нового плагина для collectd, который бы позволял это делать.

В разделе 2 мы обсудим, какие решения подобных задач уже существуют, а также какие библиотеки нам удобно использовать в проекте. В разделе 3 посмотрим на техническую реализацию плагина. Затем в разделе 4 рассмотрим процесс тестирования приложения. А в разделе 5 представим результаты работы от лица пользователя. Наконец в разделе 6 подведём итоги проекта.

2 Обзор литературы

Существует множество библиотек для взаимодействия с Telegram Bot API [9]. Тем не менее, в официальном списке Телеграма [8] нет ни одной рекомендации для языка C. Это связано с тем, что обычно для таких целей используют более высокоуровневые языки программирования, например, Python. На GitHub можно найти несколько непопулярных реализаций на C, например, *telebot* [7]. Эта библиотека позволяет довольно легко получать сообщения, извлекать из них текст и отправлять ответное сообщение. Но из-за непопулярности, дополнительных зависимостей внутри подобных библиотек, а также избыточного функционала, было решено реализовать все нужные функции по работе с Телеграмом самостоятельно.

Так как задача состоит в написании модуля к уже существующей системе collectd [1], необходимо следовать её архитектурным правилам [2]. Все модули в collectd, которые пользователь указал в конфигурации, подключаются как динамические библиотеки на C. Каждый модуль должен реализовывать какие-то из следующих функций: *Configuration, Initialization, Read, Write, Flush, Shutdown, Log, Notification*. Нас интересует последнее. Возьмём за пример существующий модуль *notify_email* [4], который отправляет нотификации по почте. Он реализован довольно аккуратно и предлагает минимум функционала. Всё, что не связано с отправкой по почте через smtp, может помочь в реализации нашего модуля отправки через Телеграм – *notify_telegram*.

Для взаимодействия с сервером Телеграма по HTTPS будем использовать популярную библиотеку *libcurl* [5]. Она предоставляет простой интерфейс, но требует аккуратной инициализации и очистки. Эта библиотека уже используется во многих модулях collectd, поэтому зависимость от неё не будет проблемой.

Для парсинга json-ответов от Телеграма используем небольшую библиотеку *yajl* [10]. Это потоковый парсер, а значит он использует мало памяти и работает быстро. Как и *libcurl*, библиотека *yajl* уже используется в других модулях collectd, поэтому выбор пал на неё.

Для возможности общения бота и сервера через протокол вебхуков подойдёт удобная библиотека *libmicrohttpd* [6]. Она позволяет разворачивать свой HTTP-сервер и пошагово обрабатывать входящие запросы, что нам и нужно. Эта библиотека, как и предыдущие, много раз использовалась в других плагинах collectd.

3 Реализация

Подробно разберём, как в коде реализована заявленная функциональность.

3.1 Зависимости и конфиг

Таблица 3.1: Таблица со всеми параметрами конфигурации плагина notify_telegram

Название	**Дефолтное значение**	**Описание**
BotToken	–	Токен телеграм-бота
RecipientChatID	–	ID чата в Телеграм, которому необходимо присылать нотификации
ProxyURL	*https://api.telegram.org/bot*	URL прокси-сервера Телеграм
WebhookHost	–	Хост, которому Телеграм будет присылать обновления по вебхукам
WebhookPort	*“443“*	Порт хоста, по которому Телеграм будет присылать обновления по вебхукам
WebhookURL	*““*	URL (кроме хоста), по которому Телеграм будет присылать обновления по вебхукам
MHDDaemonHost	–	Хост, на котором будет запущен http-сервер для получения обновлений по вебхукам
MHDDaemonPort	–	Порт хоста, на котором будет запущен http-сервер для получения обновлений по вебхукам
DisableGettingUpdates	*false*	Не получать и не обрабатывать сообщения от Телеграм

Наш плагин зависит от наличия внешних библиотек: *libcurl*, *yajl*, *libmicrohttpd*. Учтём это, добавив соответствующую проверку в файл *configure.ac* (используется в autoconf). Также добавим необходимые флаги для компилятора и компоновщика в файл *Makefile.am*. Теперь сборка проекта будет проходить корректно.

Следующим шагом добавим шаблон конфига для нашего плагина в файл *collectd.conf.in*. Внутри основного файла нового плагина (*notify_telegram.c*) все переменные со значениями конфигурации будем хранить в отдельной статической структуре *plugin_config*. Осталось только научиться заполнять эти переменные. Для этого создадим и зарегистрируем статическую функцию *notify_telegram_notification*, в которую система будет передавать ключ и

значение из конфига. Будем смотреть на ключи и заполнять соответствующие переменные в *plugin_ config*. Это упрощённая схема парсинга конфига, в нашем случае она подойдёт. Большинство переменных – строковые, будем сохранять их с помощью функции *strdup*. Но, т.к. эта функция выделяет новую память на куче, нужно не забыть очищать эти переменные при правильном завершении работы плагина. Для получения целочисленного номера порта из строки воспользуемся стандартной функцией s*trtol*. Для сохранения списка получателей сообщений будем поддерживать количество получателей и расширять список с помощью *realloc*.

Смысл каждого параметра в конфигурации плагина описан в Таблице 3.1.

3.2 Вспомогательные функции

В этом разделе рассмотрим важные вспомогательные функции, которые часто переиспользуются в других местах кода.

3.2.1 Запросы к Telegram Bot API

Для отправки запросов к Telegram Bot API была выделена отдельная функция - *telegram_ bot_ api_ send_ request*. Она принимает в качестве аргументов: ссылку на буффер, в который нужно сохранить ответ на запрос; URL, по которому нужно отправить запрос; параметры, которые нужно передать в запросе. Функция использует библиотеку *libcurl*.

Сначала формируется полный URL следующим образом: адрес прокси-сервера + токен бота + URL из аргемента функции. Если прокси-сервер не указан в конфиге, берётся стандартный URL Телеграма - *https://api.telegram.org/bot*. Затем создаётся *handle* - объект, отвечающий за запрос. К нему добавляются опции: *CURLOPT_ POSTFIELDS* - параметры из аргумента функции; *CURLOPT_ URL* - ранее построенный полный URL; *CURLOPT_ WRITEFUNCTION* - функция-обработчик ответа на запрос (в нашем случае используем *tg_ curl_ write_ callback*, в которой просто записываем все байты в один буффер); *CURLOPT_ WRITEDATA* - ссылка на буффер ответа из аргумента функции. Когда *handle* готов, вызываем библиотечную функцию *curl_ easy_ perform(handle)*, которая осуществляет http-запрос и возвращает код ответа. Когда запрос выполнен, мы очищаем *handle* с помощью библиотечной функции *curl_ easy_ cleanup(handle)*, логируем ответ в дебажном режиме и возвращаем код ответа наружу.

Отдельно стоит отметить, что запросы к Telegram Bot API через *curl_ easy_ perform* мы осуществляем под мьютексом. Это важно, т.к. collectd работает в многопоточном режиме

и нужно это учитывать. Попытка одновременного обращения к API из разных потоков может привести к сбою со стороны клиента или со стороны сервера.

Для рассылки сообщений заданным пользователям была реализована отдельная функция - *telegram_bot_api_send_message*. Она принимает в качестве аргументов: ссылку на буффер, в который нужно сохранить ответ на запрос; параметры, которые нужно передать в запросе; ссылку на сообщение; ссылку на массив, состоящий из *chat_id*; количество чатов, в которых нужно послать сообщение. Функция в целом похожа на предыдущую и тоже использует библиотеку *libcurl*.

Сначала формируется полный URL по аналогии с предыдущей функцией. Затем создаётся *handle* и выставляются все те же параметры, кроме *CURLOPT_POSTFIELDS*. Затем в цикле перебираются все *chat_id*. Для каждого формируются параметры запроса *sendMessage*: выставляется *chat_id* и строится *text*. Затем получившийся запрос отправляется под мьютексом через *curl_easy_perform(handle)*, как и в предыдущей функции. В отличии от *telegram_bot_api_send_request* здесь мы сразу не очищаем *handle*, а переиспользуем его для всех id из цикла, меняя только параметры. Когда все сообщения разосланы, мы вызываем *curl_easy_cleanup(handle)* и возвращаем последний полученный код ответа.

3.2.2 Парсинг ответов

Важная часть плагина – это парсинг получаемых от Телеграм ответов. Точнее, нас интересуют только ответы с обновлениями из чатов. Сделав простой запрос *getUpdates* в браузере, можно узнать, что ответ - это json со следующей структурой:

```
{
  "ok": true,
  "result": [
    {
      "update_id": 573531097,
      "message": {
        "message_id": 311,
        "from": {
          "id": 841760208,
          "is_bot": false,
          "first_name": "Name",
          "last_name": "Surname",
```

```
            "username": "nick123",
            "language_code": "ru",
            "is_premium": true
          },
          "chat": {
            "id": 841760208,
            "first_name": "Name",
            "last_name": "Surname",
            "username": "nick123",
            "type": "private"
          },
          "date": 1715563849,
          "text": "hello world!"
        }
      }
    ]
}
```

Из всех этих данных нас интересует только три значения: “ok“ – успешен ли запрос или нет; “result/update_id“ – нужно найти максимальное значение, чтобы потом обновить последнее прочитанное; “result/message/chat/id“ – id чата, по которому будем присылать ответ. Нужно научиться парсить эти три значения для всех обновлений. Для этого воспользуемся библиотекой *yajl*.

Вспомогательная функция, которая будет уметь парсить – *telegram_bot_api_parse_response*. Она примет в качестве аргументов: ссылку на буффер, в котором лежит json, который нужно распарсить; ссылка на структуру *parse_context*, которая будет выступать в роли внутреннего состояния.

```
struct parse_context_t {
  int depth;
  bool inside_key_ok;
  bool inside_key_update_id;
  bool inside_key_message;
  bool inside_key_message_chat;
  bool inside_key_message_chat_id;
```

```
    bool ok;
    char *max_update_id;
    int chat_id_count;
    char *chat_id[MAX_INPUT_MESSAGES_COUNT];
};
```

Внутри функции мы определяем массив *ycallbacks*, состоящий из функций-обработчиков разных типов данных (bool, number, string, map и т.д.). Эти функции будут вызываться по мере потокового парсинга данных. Далее мы аллоцируем хендлер для библиотеки при помощи функции *yajl_alloc*. Наконец, мы запускаем функцию *yajl_parse*, которая и произведёт парсинг. Если произошли ошибки парсинга, логируем их.

Идея парсинга в нашем случае проста. По мере движения вглубь и наружу json будем обновлять переменную глубины *depth*. Также будем отмечать, находимся ли мы сейчас внутри нужного ключа нужной глубины. Таким образом, например, чтобы понять, что мы сейчас парсим интересуемое значение "result/message/chat/id", должны быть выставлены флаги *inside_key_message, inside_key_message_chat, inside_key_message_chat_id*. В таком случае записываем значение в конец массива *chat_id* и увеличиваем *chat_id_count* на единицу. Или например, если мы находимся на глубине 1 и ключ это "ok", то выставляем флаг *inside_key_ok*. Таким образом, аккуратно описав всю логику в обработчиках yajl, мы получаем рабочий парсер.

Стоит также учесть, что при обработке обновлений через вебхуки json выглядит немного по-другому. Там нет верхнеуровневых ключей "ok" и "result". Нам передают сразу значение внутри "result". Тем не менее, наш парсер всё ещё будет корректно работать, надо всего лишь заменить начальную глубину *depth* с 0 на 1.

3.3 Инициализация и корректное завершение

Инициализация плагина происходит в функции *notify_telegram_init*. Там мы задаём глобальное состояние библиотеки *libcurl* с поддержкой SSL при помощи метода *curl_global_init (CURL_GLOBAL_SSL)*. Далее, в зависимости от того, будет бот взаимодействовать через вебхуки или через long polling, мы посылаем по Telegram Bot API запрос *setWebhook* или *deleteWebhook* соответственно. Запросы отправляем через вспомогательную функцию *telegram_bot_api_send_request*(см. Раздел 3.2.1).

В случае включенных вебхуков в запрос *setWebhook* нужно передать URL, по которому Телеграм будет присылать обновления. Этот URL можно построить, сложив *WebhookHost*,

WebhookPort и *WebhookURL* из конфига. Также на стадии инициализации необходимо запустить http-сервер из библиотеки *libmicrohttpd*. Для этого вызываем функцию *telegram_ start_ daemon*. Эту функцию реализуем в двух вариантах и с помощью директивы препроцессора *ifdef* в коде остаётся только одна версия, в зависимости от установленной версии библиотеки *libmicrohttpd*. Для версий ниже 0.9.0 просто вызывается функция *MHD_ start_ daemon*, которая запускает сервер. В неё передаём собственный логгер, который будет выводить ошибки в STDOUT, а также функцию-хендлер для обработки обновлений от Телеграм по вебхукам (подробнее про неё смотреть в Разделе 3.6). Для более новых версий библиотеки воспользуемся возможностью самостоятельно создавать сокет и передавать его внутрь *MHD_ start_ daemon*. Это позволяет выбрать для сервера произвольный хост и порт. При этом важным моментом является выбор протокола IPv4, т.к. Bot API Телеграма не поддерживает IPv6. Сокет создаём посредством последовательных системных вызовов *getaddrinfo, socket, setsockopt, bind, listen*. Здесь важно обработать все возможные ошибки и залогировать их, чтобы проще было разобраться при проблемах с сетью.

В случае отключённых вебхуков в конце инициализации мы сразу вызываем функцию *notify_ telegram_ read*, чтобы прочитать новую пачку сообщений боту через long polling и сразу ответить, не дожидаясь таймера.

При завершении работы плагина вызывается функция *notify_ telegram_ shutdown*. В ней мы очищаем внутреннее состояние библиотеки *libcurl* при помощи метода *curl_ global_ cleanup()*. Также здесь мы останавливаем фоновый поток с htpp-сервером вебхуков, если он был, с помощью метода *MHD_ stop_ daemon*. В конце не забываем освободить динамически выделенную память под переменные из конфига плагина.

3.4 Нотификации

Основная функция в плагине – это отправка нотификаций (уведомлений, алертов) через Телеграм. Нотификация в collectd - сущность, содержащая информацию о том, что конкретная метрика на конкретном хосте вышла за рамки разумных значений. Нотификации появились начиная с версии collectd-4.3 и пока что не полностью поддерживаются в предрелизной версии collectd-6.0. Нотификации могут генерировать некоторые плагины. Основной же источник нотификаций – *Thresholds*. В конфиге collectd можно настроить нижние и верхние пределы произвольных метрик.

Реализуем функцию *notify_ telegram_ notification* и зарегистрируем её в качестве обработчика нотификаций. Нотификация в этот обработчик подаётся в виде структуры данных

notification_t. Возьмём из неё все значения (серьёзность, хост, плагин, инсталляция плагина, тип, инсталляция типа и сообщение) и запишем их все в одну строку, разделяя знаком переноса строки. Для этого воспользуемся своей небольшой функцией *buffer_append*, которая выполняет добавление строки в конец буффера и сдвигает указатель при помощи арифметики указателей. Наконец, воспользуемся вспомогательной функцией *telegram_bot_api_send_message* (см. Раздел 3.2.1), чтобы разослать полученную строку с нотификацией всем получателям (чатам). В качестве *parse_mode* для Telegram Bot API укажем HTML, т.к. в нём меньше символов имеют специальные значения. Тем самым меньше шансов неправильно отформатировать строку. А заголовок нотификации можно выделить жирным, обернув его в тэг **:

```
buffer_append(&buf_ptr, &buf_len, "<b>Notification:</b>\nseverity", severity);
buffer_append(&buf_ptr, &buf_len, "host", n->host);
buffer_append(&buf_ptr, &buf_len, "plugin", n->plugin);
buffer_append(&buf_ptr, &buf_len, "plugin_instance", n->plugin_instance);
buffer_append(&buf_ptr, &buf_len, "type", n->type);
buffer_append(&buf_ptr, &buf_len, "type_instance", n->type_instance);
buffer_append(&buf_ptr, &buf_len, "message", n->message);
```

3.5 Long polling

По умолчанию плагин будет получать обновления от сервера Телеграм через long polling. Это техника, при которой программа регулярно с заданной частотой посылает запрос на сервер, чтобы узнать, появилась ли новая информация. В случае Telegram Bot API используется запрос *getUpdates*.

Для целей поллинга сделаем функцию *notify_telegram_read* и зарегистрируем её в качестве read-обработчика. Таким образом мы подключим функцию к стандартному шедулеру collectd, который будет запускать эту функцию через равные промежутки времени (этот параметр настраивается в конфиге collectd). В начале функции проверим, хотим ли мы использовать long polling. Если включена опция *DisableGettingUpdates* или включены вебхуки - то не хотим. Далее воспользуемся вспомогательной функцией *telegram_bot_api_send_request* (см. Раздел 3.2.1), чтобы отправить запрос *getUpdates* и сохранить ответ в буффер. В качестве параметра запроса передадим лимит на количество сообщений, которое мы готовы обработать за раз. Этот лимит хранится в константной переменной *MAX_INPUT_MESSAGES_COUNT*. Полученный ответ теперь надо распарсить, а именно - найти все chat_id внутри него. Для

этого воспользуемся другой вспомогательной функцией - *telegram_bot_api_parse_response* (см. Раздел 3.2.2). Далее проверяем, что полученное сообщение корректно (содержит статус “ok“) и содержит ненулевое количество id чатов. Если так, отправляем каждому получателю ответное сообщение в Телеграм, которое будет содержать конфигурацию плагина с заполненным полем *RecipientChatID*. Таким образом пользователю не придётся находить свой номер чата сторонними средствами. Отправку ответных сообщений реализуем при помощи вспомогательного метода *telegram_bot_api_send_message* (см. Раздел 3.2.1). Параметр *parse_mode* выберем *“MarkdownV2“*. Это позволит красиво отформатировать сообщение: конфиг будет выделен как блок кода на языке *XML* (смотреть Раздел 5.1). Когда все сообщения отправлены, остаётся послать ещё один запрос в Telegram Bot API - *getUpdates* вместе с установленным параметром *offset*. В качестве *offset* нужно указать номер максимального прочитанного ранее сообщения + 1, чтобы Телеграм знал, что мы их обработали, и их больше не нужно нам присылать. Этот максимальный номер мы также уже посчитали во время парсинга. Таким образом, в конце функции мы должны ещё раз воспользоваться вспомогательной функцией *telegram_bot_api_send_request*.

3.6 Вебхуки

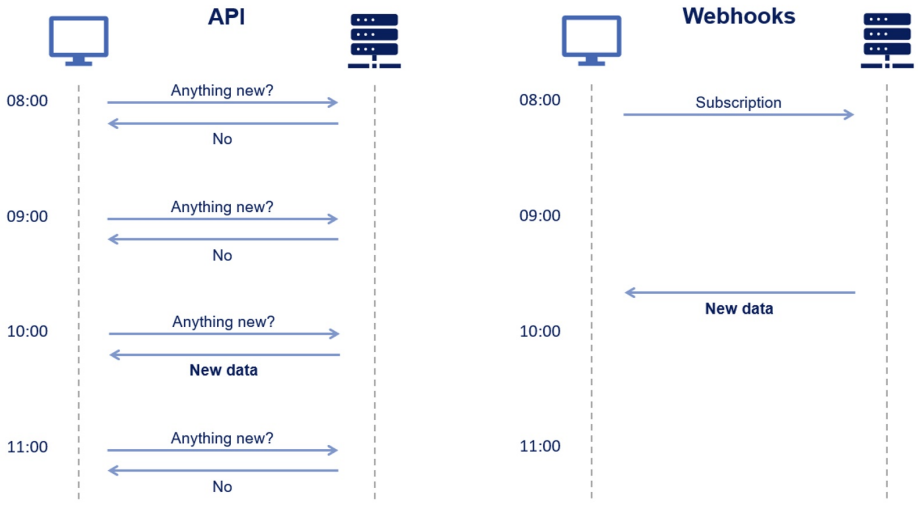

Рис. 3.1: Схема, объясняющая принцип работы Вебхуков. Оригинал: ссылка

Вебхуки (*Webhook*) - это метод (протокол) получения информации от сервера к клиенту через обратные http-запросы. То есть клиент разворачивает свой собственный сервер и передаёт исходному серверу URL, по которому тот может отправлять запросы. После чего сервер начинает отправлять запросы клиенту по этому URL, когда происходят какие-то

события. Схематично принцип работы вебхуков объясняется на Рисунке 3.1

Telegram Bot API позволяет взаимодействовать с ним через вебхуки. В Разлеле 3.3 мы разобрались, как инициализировать вебхук-соединение. Далее Телеграм будем присылать нам обновления каждый раз, когда боту кто-то будет писать сообщение. Такой подход позволяет избежать постоянных запросов от клиента к серверу, а также значительно снизить время ответа бота на сообщение, но требует более сложной настройки и в некоторых случаях может быть небезопасен. Последний недостаток можно компенсировать, если использовать Local Bot API Server. Сравнение методов взаимодействия с Telegram Bot API для нашей задачи приведены в Таблице 3.2.

Таблица 3.2: Таблица сравнения методов взаимодействия с Telegram Bot API.

Признак сравнения	**Long polling**	**Webhook**	**Webhook + Local Bot API Server**
Легкость настройки	легко	сложнее	ещё сложнее
Безопасность	безопасно	менее безопасно	безопасно
Скорость ответа	медленно	быстро	быстро

Важную роль в реализации играет функция-хендлер *telegram_mhd_handler*, вызываемая каждый раз, когда приходит новое сообщение. В этой функции мы сначала проверяем, что пришёл POST-запрос и именно на тот URL, который мы передавали при инициализации плагина в *setWebhook*. Далее, если это первый вызов хендлера, мы инициализируем буффер под сообщение и возвращаем *MHD_YES*. Иначе мы заполняем буффер данными из тела запроса, пока оно не окажется пустым. Это связано с тем, что при получении нового запроса библиотека *libmicrohttpd* вызывает функцию-хендлер несколько раз: один раз, чтобы обработать только заголовки, и далее несколько раз, чтобы небольшими кусками обработать тело запроса. Когда сообщение получено полностью, используем нашу вспомогательную функцию *telegram_bot_api_parse_response* для парсинга *chat_id* (см. Раздел 3.2.2). Далее, если удалось распарсить чаты, шлём в них сообщения с инструкциями через вспомогательную функцию *telegram_bot_api_send_message* (см. Раздел 3.2.1). Также следим, чтобы на все сетевые запросы получать в ответ *200 OK*. В конце концов, когда все сообщения доставлены, возвращаем ответ на первоначальный запрос - *200 OK* с пустым телом. Для этого используем библиотечные функции *MHD_create_response_empty* и *MHD_queue_response*.

4 Тестирование

4.1 Бот

Рис. 4.1: Скриншот с информацией о BotFather.

Для тестирования нам понадобится создать бота в Телеграм. Все боты создаются через личные сообщения с главным ботом Телеграм – BotFather. Информация об этом боте есть на Рисунке 4.1. Надо написать ему команду */newbot*. Далее следуем его инструкциям: указываем название бота, описание. По желанию выбираем для него картинку. После этого бот создан и нам выдают токен. Этот токен необходимо будет указать в конфигурации нашего плагина *notify_telegram*, чтобы сообщения приходили от имени только что созданного бота.

4.2 Плагин

Тестирование плагина производится вручную на локальном устройстве. Для этого необходимо предварительно установить все нужные библиотеки. Затем запустить сборку и установку плагина. После этого необходимо настроить конфигурацию и запустить collectd в тестовом режиме, в котором плагины запускаются и сразу останавливаются. Все эти действия можно сжать до одной команды.

```
sudo rm -rf /opt/collectd && ./build.sh
&& ./configure --enable-notify-telegram
```

```
&& make && sudo make install
&& sudo chown -R username:username /opt/collectd
&& cp ~/tmp/collectd.conf /opt/collectd/etc/collectd.conf
&& /opt/collectd/sbin/collectd -t
```

Если на этом этапе проблем не выявлено, можно запускать collectd в рабочем режиме.

```
/opt/collectd/sbin/collectd -f
```

После чего тестирование производится путём написания личных и групповых сообщений боту в мессенджере Телеграм. В ответ ожидается стандартная инструкция. Также ожидается, что будут приходить сообщения с сработавшими нотификациями collectd.

Для того, чтобы всегда гарантированно получать какие-то нотификации, в конфигурации collectd указывается заведомо низкий или высокий порог на какую-то из метрик системы. Например:

```
LoadPlugin "threshold"
<Plugin threshold>
        <Plugin "interface">
                Instance "eth0"
                <Type "if_octets">
                        FailureMin 5
                        FailureMax 6
                </Type>
        </Plugin>
</Plugin>
```

5 Результаты

Рассмотрим, как выглядят результаты работы нашего плагина с точки зрения пользователя. Какие сообщения он видит и как может их использовать.

5.1 Конфигурация

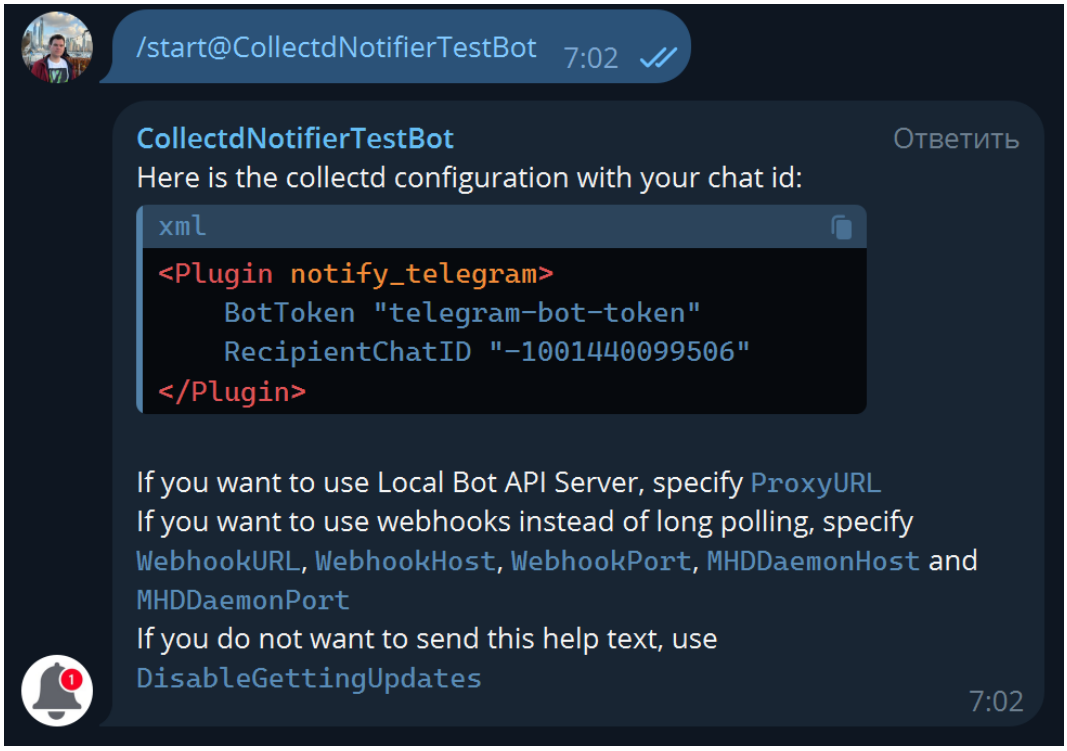

Рис. 5.1: Скриншот с примером ответа бота на сообщения. Бот предоставляет заполненную конфигурацию.

В конфигурации к плагину пользователь должен указать токен для телеграм-бота (**BotToken**). Если требуется использовать прокси-сервер, нужно указать его URL (**ProxyURL**). Далее для каждого чата (группового или личного), в который требуется присылать уведомления, нужно указать его ID (**RecipientChatID**).

Чтобы не узнавать сторонними способами ID для каждого из чатов, можно написать боту любое сообщение, затем запустить collectd, и бот ответит вам заполненной конфигурацией, где уже указан ID. Пример ответа изображён на Рисунке 5.1. Это особенно полезно, когда хочется добавить уведомления для инженеров, которые непосредственно collectd не настраивают, но отвечают за часть инфраструктуры. В таком случае очень удобно будет переслать прямо в Телеграме это приветственное сообщение бота тому, кто правит конфиг. Таким образом, от инженера-электронщика не потребуется дополнительных знаний.

Также у пользователя есть возможность настроить такие параметры, как **WebhookHost**,

WebhookPort, **WebhookURL**, **MHDDaemonHost**, **MHDDaemonPort**, чтобы бот начал использовать вебхуки, тем самым ускорив выдачу ответа. Если ответы бота больше не нужны, их можно отключить, указав опцию **DisableGettingUpdates**.

5.2 Уведомления

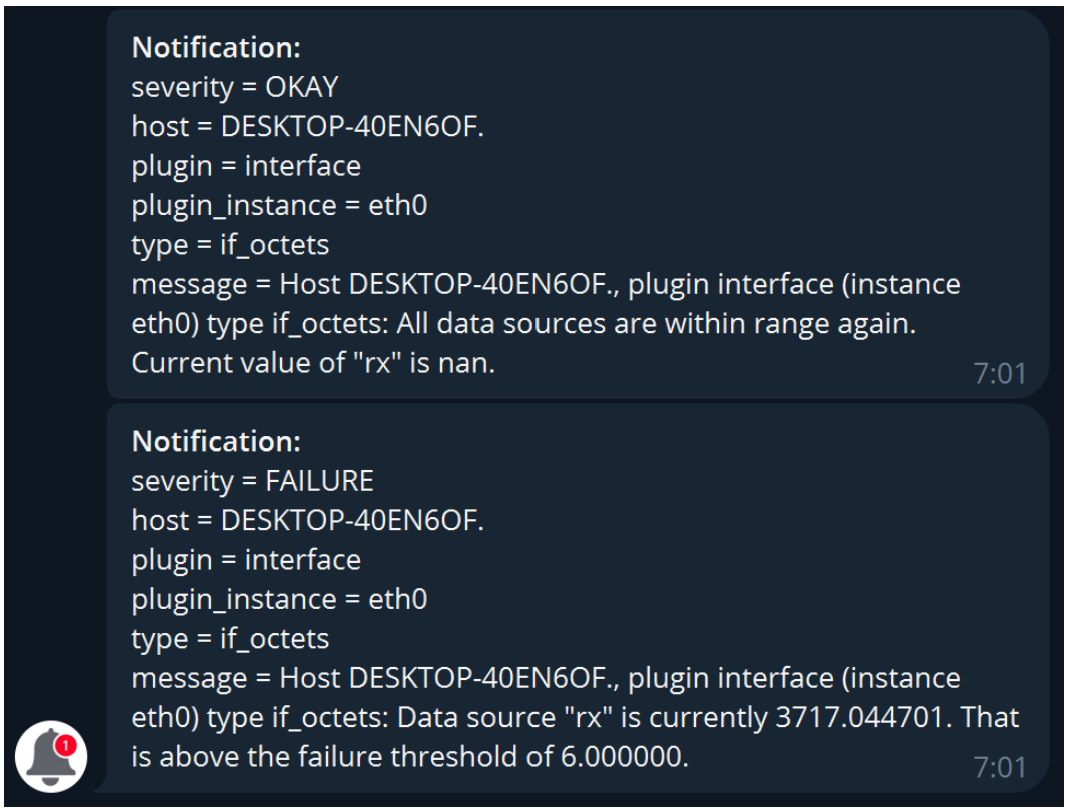

Рис. 5.2: Скриншот с примером нотификации от бота. Бот присылает актуальные уведомления во все указанные пользователем чаты.

Во время работы сервера collectd все генерируемые нотификации обрабатываются нашим плагином и отправляются в виде сообщений в Телеграме. Пример нотификации изображён на Рисунке 5.2. В сообщении содержится информация о типе уведомления, хосте, плагине, инсталляции, а также краткий комментарий о состоянии метрики.

6 Заключение

Поставленная цель была достигнута. Успешно удалось реализовать и протестировать плагин оповещений через мессенджер Telegram для сервера мониторинга collectd. Код плагина был оформлен в два пулл-реквеста (отдельно для версии 5 и версии 6) в оригинальный репозиторий collectd и отдан на ревью опытным людям из проекта. Также были учтены те правки, которые уже успели появиться там.

Полученный в результате работы код будет полезен для многих пользователей collectd, которые хотят оперативно реагировать на нотификации в своих системах. Также у плагина есть потенциал – его можно дорабатывать и расширять функционал того, что умеет бот.

Pull request для collectd-5: https://github.com/collectd/collectd/pull/4303

Pull request для collectd-6: https://github.com/collectd/collectd/pull/4311

Список литературы

[1] Collectd. *GitHub repository*. URL: https://github.com/collectd/collectd.

[2] Collectd. *Plugin architecture*. URL: https://github.com/collectd/collectd/wiki/Plugin-architecture.

[3] Collectd. *Website*. URL: https://www.collectd.org/.

[4] Oleg King и Florian octo Forster. *Plugin Notify Email*. URL: https://github.com/collectd/collectd/wiki/Plugin-Notify-Email.

[5] Libcurl. *Website*. URL: https://curl.se/libcurl/c/.

[6] Libmicrohttpd. *Website*. URL: https://www.gnu.org/software/libmicrohttpd/.

[7] Elmurod Talipov. *Telegram Bot in C*. URL: https://github.com/smartnode/telebot.

[8] Telegram. *Bot Library Examples*. URL: https://core.telegram.org/bots/samples.

[9] Telegram. *Telegram Bot API*. URL: https://core.telegram.org/bots/api.

[10] YAJL. *Website*. URL: https://lloyd.github.io/yajl/.