Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 20.Б10-мм

Тагильцев Семен Вадимович

Pacemaker: управление ресурсами через плагины

Отчёт по учебной практике

в форме «Производственное задание»

Научный руководитель:
ст. преп. каф. ИАС, К. К. Смирнов

Санкт-Петербург
2023

Оглавление

1. Введение

В современном мире многие большие IT-компании имеют сервера с критически важными базами данных или бизнес-приложениями, которые должны быть доступны практически всегда. Для выполнения этой задачи существуют отказоустойчивые кластеры [6]. IT-компания YADRO[1], ведущий российский разработчик и производитель высокоэффективных серверов и систем хранения данных корпоративного класса, не является исключением. Отсутствие High-Availability кластера приводит к тому, что из-за выхода из строя одного из серверов становятся недоступными и приложения, которые были на нём запущены. Такая проблема может привести к потерям данных, клиентов, а вследствие чего и прибыли. Отказоустойчивая кластеризация позволяет мониторить состояние своих серверов и своевременно обнаруживать неисправности, а также запускать процессы борьбы с ними без вмешательства системных администраторов, тем самым минимизируя риск сбоев на стороне пользователя. Зачастую это достигается за счет разворачивания новых серверов, которые в свою очередь являются копиями неисправных, или соответствующей конфигурации имеющихся [1].

Одним из основных компонентов отказоустойчивых систем компании ClusterLabs[2] является менеджер ресурсов с открытым исходным кодом pacemaker [2]. Ресурсы кластера управляются при помощи агентов, которые представляют собой внешние скрипты, запускаемые при помощи стандартного для Unix систем метода – fork-exec. Что касается компании YADRO, то в своих высоко-доступных кластерах они используют большое количество ресурсов, которые зачастую лишь пишут заранее подготовленные данные в файлы. Иначе говоря, подготовка и создание нового процесса для таких простых действий выглядит излишней.

Возможным усовершенствованием pacemaker, а также и решением данной проблемы, является добавление поддержки разделяемых биб-

[1]Компания YADRO: https://yadro.com/ (online; accessed: 2023-01-03).

[2]Компания ClusterLabs: https://clusterlabs.org (online; accessed: 2022-01-03).

лиотек как альтернативы внешним скриптам. Такие библиотеки можно будет динамически подгружать в код, не порождая при этом дополнительных процессов, что должно положительно сказаться на производительности. При этом важно обеспечить обратную совместимость: в случае если не получилось подгрузить ресурс динамически, инициировать стандартный механизм запуска.

Также взаимодействие с динамическими библиотеками может быть вынесено в отдельный класс агентов. Вместо того чтобы сохранять обратную совместимость с уже имеющимся классом, можно разработать и реализовать свой класс агентов. Это позволит гибко настроить его под себя, тем сам отказаться от избыточных, а порой даже мешающих аспектов старых агентов.

2. Постановка задачи

Целью работы является разработка системы плагинов. Для достижения данной цели были поставлены следующие задачи.
Осенний семестр:

1. погрузиться в предметную область: развернуть простой HA кластер на двух виртуальных машинах;
2. разработать и реализовать пример плагина с шаблоном;
3. разработать и реализовать системы загрузки динамических библиотек;
4. провести замеры производительности.

Весенний семестр:

1. изучить различные классы агентов, представленные в pacemaker;
2. разработать и реализовать новый класс агента;
3. разработать и реализовать пример плагина с шаблоном;
4. провести замеры производительности.

3. Обзор

3.1. Pacemaker

3.1.1. OCF класс

Вызов агентов в стандартной конфигурации pacemaker осуществляется при помощи характерного метода для Unix систем – fork-exec.

В связи с тем, что агенты в оригинальной сборке pacemaker запускаются в отдельных дочерних процессах, то для передачи информации между процессами используются каналы. Стоит отметить, что в дочернем процессе подменяются стандартные файловые дескрипторы на соответствующие каналов, тем самым позволяя агенту не задумываться о том, с какими дескрипторами работать.

Так как работа с разделяемыми библиотеками не предполагает порождения новых процессов, то и каналы нельзя оставлять. В то же время потребность в передачи информации между кодом агентов и pacemaker сохранилась. Вследствие чего было принято решение использовать временные файлы, которые живут в рамках одного потока исполнения программы и автоматически удаляются после закрытия соответствующих дескрипторов. Для данных целей хорошо подходят методы shm_open() [19] и memfd_create() [17]. Исходя из того, что метод memfd_create() не требует явного указывания имени временного файла и никаких действий связанных с удалением, выбор пал именно на него.

Другим классическим методом взаимодействия между процессами, который использует pacemaker в процессе вызова агентов ресурсов являются сигналы. При работе с плагинами отсутствует необходимость в подобного рода вещах.

По причине того, что некоторые действия агентов могут выполняться продолжительное время, которое в отдельных случаях измеряется в секундах, а порой и в минутах, в pacemaker поддерживается асинхронное выполнение действий. Кроме того, подавляющее большинство действий агентов являются асинхронными.

Из-за того, что разрабатываемая система для загрузки динамиче-

ских плагины предполагает работу с легковесными агентами, которые только лишь пишут заранее подготовленные данные в файлы, было принято решение отказаться от асинхронного вызова методов. Другими словами теперь все методы, по факту, стали синхронными.

3.1.2. Dlopen класс

Для нового класса агентов было выбрано рабочее название – dlopen (аналогично названию метода, который отвечает за загрузку динамических библиотек).

В pacemaker представлены такие классы, как OCF, LSB [10], nagios [11], upstart [13], systemd [12]. Различные классы агентов по-разному взаимодействуют с агентами. Например, инициализирующие скрипты классов systemd и upstart вызываются при помощи технологии D-Bus [3].

После изучения исходных кодов различных классов агентов, можно сделать вывод, что новый класс должен поддерживать следующие методы:

- метод, который возвращает имена всех агентов данного класса;
- метод, который проверяет наличие данного агента;
- метод, который используется при подготовке агента к исполнению;
- метод, который преобразует коды возврата к OCF кодам;
- метод, который отвечает за вызов агентов;

Стоит отметить, что не все из этих методов являются обязательными. Новый класс агентов смог бы вызывать агенты имея только последний метод.

Общая концепция вызова агентов схожа с той, что описана выше для OCF класса. Отличительной особенностью является работа с передачей информации между агентом и pacemaker. Из-за своей простоты,

в частности, отсутствия взаимодействия с временными файлами, было принято решение передавать в агент указатель на строку.

3.2. Агент

3.2.1. OCF агент

Для написания агентов для pacemaker существует стандарт, который называется Open Cluster Framework (OCF) [9]. Кратко говоря, агенты представляют собой исполняемые файлы, которые соответствуют основным положениям изложенным в документации стандарта OCF.

Классические агенты, которые входят в стандартную конфигурацию pacemaker, являются исполняемыми файлами с набором обязательных методов: start, stop, matadata и monitor [14], поэтому будущий плагин должен реализовывать эти методы. Особенностью является то, что плагин должен быть не только разделяемым объектом, но и исполняемым файлом. Это связано с тем, что не только непосредственно pacemaker вызывает агента, а также и pcs – интерфейс командной строки для pacemaker, например, для получения метаданных. Известным примером исполняемого файла, который является динамической библиотекой, является libc.so. Следует отметить, что она написана в основном на языке программирования Си. По этой причине, а также потому, что pacemaker тоже написан на Си, было принято решение использовать данный язык программирования при реализации плагина.

Как сказано в гайде по разработке OCF агентов [8], агент ресурса получает всю информацию о конфигурации ресурса, которым он управляет, из переменных окружения. Вследствие того, что используется хитрость с заменой точки входа в программу [7], перестают работать многие методы из библиотеки libc. К счастью, метод getenv(), который используется для получения переменных окружения, не входит в этот список.

В пособии для разработчиков агентов под OCF стандартом также указано, что существует 9 специфических кодов возвратов из агента. Здесь не возникает трудности при переносе их на плоскость исполняе-

мых разделяемых объектов.

3.2.2. Dlopen агент

Благодаря тому, что реализовывается новый класс агентов и нет необходимость в обратной совместимости, то требования от агента снизились. Проще говоря, нет необходимости в том, чтоб агент был исполняемым файлом и следовал OCF стандарту. Это приносит несколько упрощений в разработке агента. В частности, агент перестает быть исполняемым файлом и становится полноценной динамической библиотекой.

Из-за отказа от хитрости с заменой точки входа в программу, используемой для того, чтоб агент был исполняемым файлом, появилась возможность линковаться с библиотеками. В частности с библиотекой glib [5], что позволило заменить способ конфигурирования агентов. Вместо неявных для программиста переменных окружения, используется хэш-таблица из glib. Хэш-таблица передается методу агента в виде параметра.

Следует учитывать то, что нет необходимости в реализации обязательных методов из OCF стандарта, а также в использовании специфических кодов возвратов. Здесь открывается огромный простор для творчества, но было принято решение сохранить их. Исходя из тех соображений, что это будет наглядно и понятно для людей, которые уже разрабатывали агенты для pacemaker.

Еще одним ощутимым плюсом того, что нет необходимости сохранять совместимость с вызовом исполняемых файлов, является отказ от логики работы с файловыми дескрипторами. Теперь агенту достаточно выделить память и присвоить необходимую строку в соответствующую переменную, вместо записи в файловый дескриптор.

4. Архитектура

Обобщая все выше сказанное можно схематично изобразить архитектуру стандартного механизма вызова агентов ресурсов pacemaker и новую систему загрузки плагинов.

4.1. Старая архитектура OCF класса

Классический pacemaker выглядит примерно следующим образом.

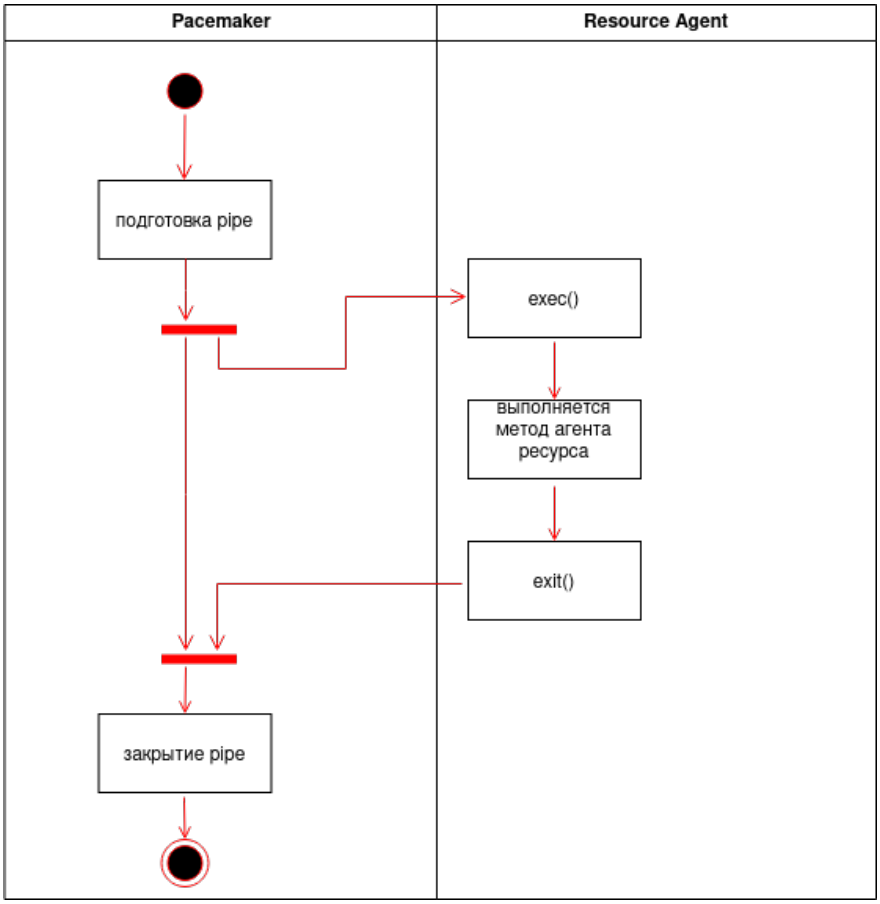

Рис. 1: Диаграмма старой архитектуры

4.2. Новая архитектура OCF класса

Что касается системы загрузки плагинов, то она имеет следующий вид. Изначально каждый агент загружается, как динамический объект. Затем в случае положительного ответа от dlopen() выполняются

действия представленные ниже. В противном случае инициализируется стандартный запуск агентов ресурсов.

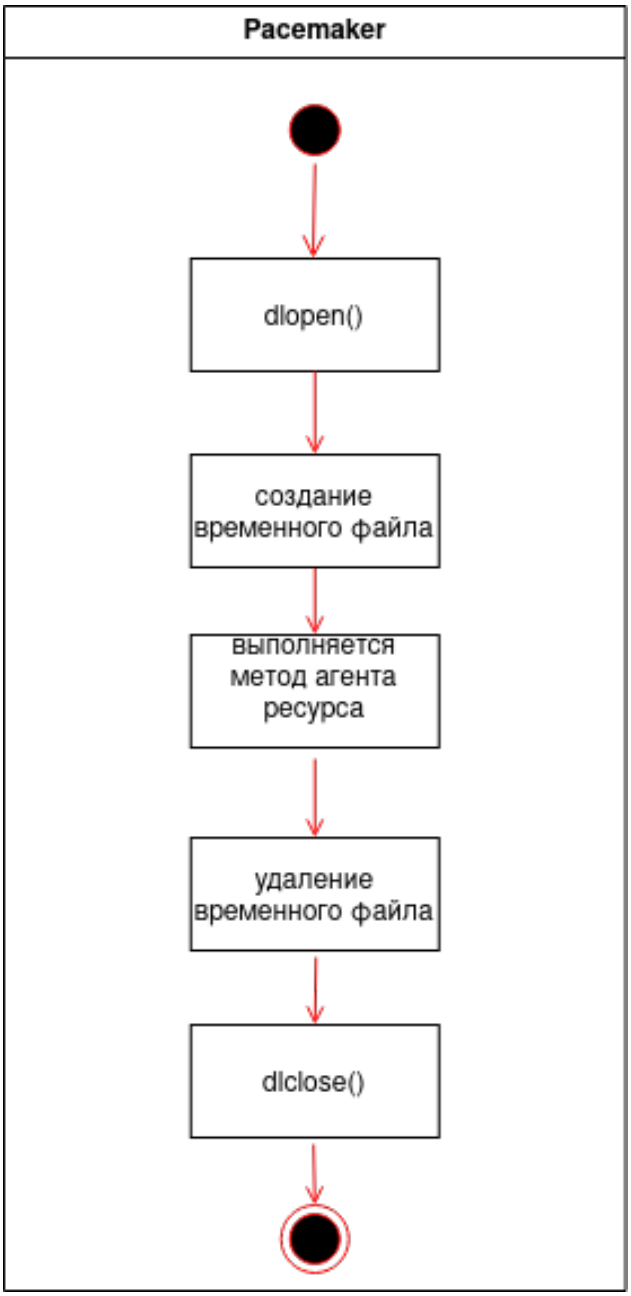

Рис. 2: Диаграмма новой архитектуры

4.3. Новая архитектура dlopen класса

Отличия динамической загрузки dlopen агентов от OCF минимальны. Во-первых, вместо подготовки переменных окружения используется хэш-таблица. Во-вторых, вместо создания временного файла и пере-

дачи файлового дескриптора указующего на него, в метод агента передается указатель на строку.

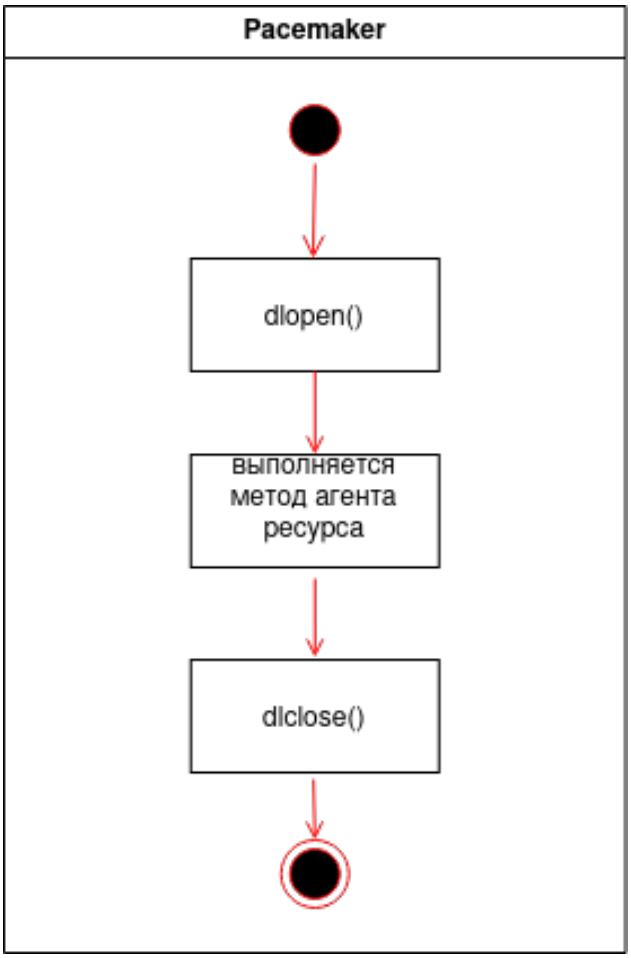

Рис. 3: Диаграмма новой архитектуры

5. Реализация

В рамках знакомства с предметной областью проекта был развернут простенький кластер по гайду от ClusterLabs. Сама же работа над курсовой велась в рамках специально подготовленного тестового стенда, который собирался из исходников при помощи Vagrant[3] и использовался для тестирования патчей к pacemaker.

5.1. Реализация системы загрузки динамических библиотек для OCF класса

Структурно загрузка динамических объектов схожа со стандартным примером из man-документации функции dlopen [16]: последовательно вызываются функции dlopen, dlsym и dlclose.

Листинг 1: Пример из документации

```
int main(int argc, char **argv) {
    void *handle;
    double (*cosine)(double);
    char *error;
    handle = dlopen ("/lib/libm.so", RTLD_LAZY);
    if (!handle) {
        fputs (dlerror(), stderr);
        exit(1);
    }
    cosine = dlsym(handle, "cos");
    if ((error = dlerror()) != NULL)  {
        fprintf (stderr, "%s\n", error);
        exit(1);
    }
    printf ("%f\n", (*cosine)(2.0));
    dlclose(handle);
}
```

[3]Vagrant: https://www.vagrantup.com/ (online; accessed: 2023-01-03).

5.1.1. Реализация взаимодействия

Изначально было необходимо подробно разобраться, каким образом взаимодействуют pacemaker и агенты ресурсов. А именно как pcmk вызывает агенты, какие перед этим подготовительные действия, как pcmk обрабатывает синхронные и асинхронные действия и многое другое.

Для передачи информации между агентом ресурса и pacemaker нужно было реализовать общение по средствам временного файла, создаваемого при помощи memfd_create.

Принцип работы такого метода довольно прост. Создается временный файл в памяти узла, в который агент записывает информацию, после чего pacemaker вычитывает ее и временный файл удаляется.

5.1.2. Технических особенностей

Система загрузки динамических библиотек обрабатывает все действия синхронно, в отличие от классического pacemaker, который имеет событийно-ориентированную архитектуру [23].

Аналогично стандартной конфигурации pacemaker, в системе загрузки плагинов устанавливаются различные флаги, код возврата агента ресурса, а также передаются данные через модель.

Невозможно не упомянуть про трудность, которая возникала в процессе реализации системы динамической загрузки библиотек и была связана с наиболее значимой разновидностью действий – повторяющиеся. Это действия которые автоматически выполняются через заданные промежутки времени. Одним из примеров таких действия является monitor, который следит за состоянием ресурса. Проблема заключалась в том, что по истечению времени запланированное действие не вызывалось, хотя в логах не было никаких ошибок. Выяснилось, что после каждого вызова действия необходимо было перепланировать его.

5.2. Реализация системы загрузки динамических библиотек для dlopen класса

Прежде всего необходимо было изучить существующие в pacemaker классы агентов, в частности: OCF, LSB и nagios. После чего было реализовано 7 функций в новом классе агентов.

Поиском по именам классов агентов, а также по названию функций, в исходном коде агентов были найдены места, где они используются. После чего в эти места оставалось лишь добавить вызов методов из нового класса. Например, на Листинг 2 показано, как стала выглядеть функция, отвечающая за вызов исполнителя действия в зависимости от стандарта. Здесь можно видеть добавленные строчки для обработки dlopen класса.

Листинг 2: Пример измененной функции

```
static int
execute_action(svc_action_t *op)
{
    if (pcmk__str_eq(op->standard, PCMK_RESOURCE_CLASS_DLOPEN,
                     pcmk__str_casei)) {
        return services__execute_dlopen(op);
    }

#if SUPPORT_UPSTART
    if (pcmk__str_eq(op->standard, PCMK_RESOURCE_CLASS_UPSTART,
                     pcmk__str_casei)) {
        return services__execute_upstart(op);
    }
#endif

#if SUPPORT_SYSTEMD
    if (pcmk__str_eq(op->standard, PCMK_RESOURCE_CLASS_SYSTEMD,
                     pcmk__str_casei)) {
        return services__execute_systemd(op);
```

```
    }
#endif

    return services__execute_file(op);
}
```

Что касается реализации функций, отвечающих за вызов агентов, то они были упрощены. Появилось разделение обработчиков для вызова metadata и всех остальных действий. Metadata, в текущей реализации, не принимает параметров и возвращает строку, в отличие от остальных методов.

Как уже говорилось ранее, переменные окружения заменились хэш-таблицей. Удобство заключается в том, что готовая хэш-таблица находится вместе с другой метаинформацией об агенте, создавать ничего не нужно. Остается только заполнить ее нужными парами ключ/значение и передать агенту.

Другим параметром, передаваемым в метод агента, является указатель на строку, вместо файлового дескриптора. Это позволяет отказаться от использования временных файлов, что положительно влияет на производительность.

Было реализовано две схожих системы загрузки динамических библиотек: одна для агентов написанных на языке Си, другая для агентов написанных на языке программирования Go. Отличие версии на Go заключается в том, что агенту передаются структуры представляющие типы из Go, вместо типов языка Си. Например, вместо хэш-таблицы из glib используется структура go_slice (представляющая слайс из Go) с тремя полями. [20]

Листинг 3: Представление Go типов

```
typedef long long int go_int;
typedef struct{const char *p; go_int len;} go_str;
typedef struct{void *arr; go_int len; go_int cap;} go_slice;
```

5.3. Реализация шаблона тестового агента для OCF класса

После изучение пособия для разработчиков агентов от ClusterLabs был создан макет агента, который соответствовал стандартной политике OCF агентов в аспекте кодов возврата, обязательного набора методов, именования глобальных переменных.

5.3.1. Реализация входа в программу

В связи с тем, что динамическая библиотека должна быть вызываемой, пришлось реализовывать хитрость с отдельной точкой входа в программу.

Как известно, у любой стандартной исполняемой программы на Си точкой входа является функция main(). В отличие от нее, тестовый агент имеет функции handler() и mmain(), которые каждая по своему обрабатывают одни и те же запросы. Первая вызывается при динамической загрузке библиотеки, вторая при вызове плагина, как исполняемого файла.

Благодаря такой реализации стало возможно скомпилировать разделяемую библиотеку таким образом, чтоб она была еще и исполняемой.

5.3.2. Реализация разбора параметров

К сожалению, реализация плагина с применением кастомной точки входа в программу накладывает некоторые ограничения. Это связано с тем, что при линковке программы отсутсвует crt0 [15], который позволяет установить переменные окружения, получить входные параметры, подготовить консоль.

Одним из таких является ограничение на передачу параметров. То есть кастомная функция, которая используется как точка входа в программу имеет сигнатуру с нулем параметров. В связи с этим разбор параметров осуществляется при помощи ручного доставания их из стека.

5.3.3. Реализация сохранения состояния агента

Для того чтобы определить в запущенном или выключенном состояние находится ресурс, а также для изменения состояния при вызове команд start и stop, был реализован метод с временным уникальным файлом.

При первом обращение к файлу в него записывается изначально подготовленное значение, которое означает что ресурс не запущен. В дальнейшем в зависимости от действий, которые вызываются состояние в файле меняется на запущенное.

Вместе с удаления ресурса из конфигурации отказоустойчивого кластера удаляется и его временный файл.

5.4. Реализация шаблона тестового агента для dlopen класса

Агент представляет собой классическую разделяемую библиотеку. Как уже говорилось ранее, были реализованы обязательные методы из стандарта OCF без ограниченней на использование методов из внешних библиотек.

Так как библиотека является классической, то никаких точек входа в программу и обработчиков здесь нет. Вместо переменных окружения конфигурация достается из хэш-таблицы. Для сообщений об ошибках используется указатель на строку, агент выделяет память и записывает текст ошибки, после чего ее можно обработать в pacemaker.

Агент написанный на языке программирования Go является практически точной копией агента на Си. Исключением является лишь то, что вместо хэш-таблиц используются слайсы.

6. Эксперимент

6.1. Эксперимент на кластере

Для проведения эксперимента был поднят тестовый отказоустойчивый кластер в конфигурации с двумя узлами и с количеством ресурсов 1000 штук. Обе виртуальные машины имеют Linux Ubuntu 20.04.5 LTS[4] в качестве ОС, а также pacemaker версии 2.1.4[5] с патчем на загрузку динамических библиотек. Разница заключается лишь в том, что на одном из узлов отключена возможность загрузки плагинов. Набор из двух узлов является минимально достаточным для построения кластера, а также позволяет варьировать аппаратные ресурсы виртуальных машин, для постановки наиболее объективного эксперимента.

Затем были произведены замеры производительности следующих сценариев:

- Поднятие узла: имитация того, что один узел вышел, и теперь ресурсы запускаются на новом.
- Отключение узла: имитация того, что произошла ошибка на узле и ресурсы корректно завершают свою работу.

Каждый из этих сценариев измерялся для разработанной системы плагинов и для классической существующей в pacemaker системы исполнения скриптов. Затраченное время вычислялось, как разница между инициализацией запуска первого ресурса и успешным запуском последнего. Сами числовые значения были взяты из логов pacemaker.

В ходе экспериментов все системы кластера работали корректно, без ошибок. Из результатов измерений приведенных в таблице можно сделать вывод, что в зависимости от конфигурации виртуальных машин значения прирост производительности сильно варьируется.

С результатами можно ознакомиться в таблице ниже. Погрешность измерений не превышает 10%.

[4]Linux Ubuntu 20.04.5 LTS: https://releases.ubuntu.com/focal/ (online; accessed: 2023-01-03).

[5]Pacemaker-2.1.4: https://github.com/ClusterLabs/pacemaker/releases/tag/Pacemaker-2.1.4 (online; accessed: 2023-01-03).

Таблица 1: Результаты эксперимента

Сценарий (конфигурация кластера)	Исполняемый файл (avg, сек.)	Плагин (avg, сек.)
Поднятие узла (1 ядро, 2Гб)	85	64
Поднятие узла (2 ядра, 2Гб)	47	39
Поднятие узла (4 ядра, 2Гб)	54	52
Отключение узла (1 ядро, 2Гб)	62	53
Отключение узла (2 ядра, 2Гб)	55	50
Отключение узла (4 ядра, 2Гб)	45	41

6.2. Эксперимент на демоне

Так как в pacemaker есть несколько «узких мест», в частности связанные с обходом графа, то результаты получаются смазанными остальной логикой приложения. Поэтому было принято решение тестировать только на демоне, который отвечает за исполнение действий: pacemaker-execd.

Ниже представлен план эксперимента:

- Запускается демон pacemaker-execd.
- Регистрируется, а затем запускается 1000 ресурсов.
- При помощи утилиты perf [18] собирается информация о вызываемых функциях.
- Строится flame-график для наглядного представления результатов.

Конфигурация узла кластера по железу такая же, как в прошлом эксперименте. Что касается ресурсов виртуальной машины, то это 2 ядра и 1 гигабайт оперативной памяти. В разделе «Приложения» расположены flame-графики [4] измерений.

При рассмотрении графов можно заметить, что время исполнения метода execute_resource_action в классическом pacemaker составляет 11.73%. В версии, где используется динамическая библиотека и сохраняется совместимость с OCF стандартом – 7.36%. В версии с новым классом dlopen – 4.17% для агентов написанных на Go и 3.36% для агентов на Си.

Другими словами, метод отвечающий за вызов действий ускорился практически в 4 раза. Также версия с новым классом dlopen работает быстрее более чем в 2 раза, чем версия, в которой сохраняется совместимость с OCF стандартом. Это достигается за счет того, что новый класс в отличие от OCF более легковесный. В нем отсутствует логика с подготовкой переменных окружения, а также действия с временными файлами.

7. Заключение

В ходе выполнения данной работы были достигнуты все поставленные цели, а именно.
Осенний семестр:

1. развернут простой HA кластер с двумя узлами для погружения в предметную область задачи;
2. реализован пример плагина с шаблоном;
3. реализована система динамической загрузки библиотек;
4. произведены замеры производительности.

Весенний семестр:

1. изучены различные классы агентов, представленные в pacemaker;
2. реализован новый класс агента;
3. реализован пример плагина с шаблоном;
4. произведены замеры производительности.

Весь исходный код реализации задачи расположен на GitHub [21, 22].

Список литературы

[1] Active-Active vs. Active-Passive High-Availability Clustering. — URL: https://www.jscape.com/blog/active-active-vs-active-passive-high-availability-cluster (online; accessed: 2023-01-03).

[2] ClusterLabs > Pacemaker. — URL: https://clusterlabs.org/pacemaker (online; accessed: 2023-01-03).

[3] D-Bus [Электронный ресурс]: Википедия. Свободная энциклопедия. — URL: https://ru.wikipedia.org/wiki/D-Bus (online; accessed: 2023-05-27).

[4] Flame Graphs. — URL: https://www.brendangregg.com/flamegraphs.html (online; accessed: 2023-05-27).

[5] Glib. — URL: https://docs.gtk.org/glib/ (online; accessed: 2023-05-27).

[6] Introduction To Clustering and High Availability. — URL: https://docs.geoserver.geo-solutions.it/edu/en/clustering/clustering/introduction.html (online; accessed: 2023-01-03).

[7] Linux executable without main() Function | Write a C Executable program without main() function. — URL: https://imvoid.wordpress.com/2013/09/18/linux-executable-without-main-function-write-a-c-executable-prog (online; accessed: 2023-01-03).

[8] The OCF Resource Agent Developer's Guide [Электронный ресурс]: GitHub. — URL: https://github.com/ClusterLabs/resource-agents/blob/main/doc/dev-guides/ra-dev-guide.asc (online; accessed: 2023-01-03).

[9] OCF Resource Agents. — URL: http://www.linux-ha.org/wiki/OCF_Resource_Agents (online; accessed: 2023-01-03).

[10] Pacemaker LSB Agent Class. — URL: https://clusterlabs.org/pacemaker/doc/2.1/Pacemaker_Explained/html/resources.html#linux-standard-base (online; accessed: 2023-05-27).

[11] Pacemaker Nagios Agent Class. — URL: https://clusterlabs.org/pacemaker/doc/2.1/Pacemaker_Explained/html/resources.html#nagios-plugins (online; accessed: 2023-05-27).

[12] Pacemaker Systemd Agent Class. — URL: https://clusterlabs.org/pacemaker/doc/2.1/Pacemaker_Explained/html/resources.html#systemd (online; accessed: 2023-05-27).

[13] Pacemaker Upstart Agent Class. — URL: https://clusterlabs.org/pacemaker/doc/2.1/Pacemaker_Explained/html/resources.html#upstart (online; accessed: 2023-05-27).

[14] Resource Agents — Pacemaker Administration [Электронный ресурс]: GitHub. — URL: https://clusterlabs.org/pacemaker/doc/2.1/Pacemaker_Administration/html/agents.html#actions (online; accessed: 2023-01-03).

[15] crt0 [Электронный ресурс]: Википедия. Свободная энциклопедия. — URL: https://ru.wikipedia.org/wiki/Crt0 (online; accessed: 2023-01-03).

[16] dlopen(3) [Электронный ресурс]: Linux manual page. — URL: https://man7.org/linux/man-pages/man3/dlopen.3.html (online; accessed: 2023-01-03).

[17] memfd_create(2) [Электронный ресурс]: Linux manual page. — URL: https://man7.org/linux/man-pages/man2/memfd_create.2.html (online; accessed: 2023-01-03).

[18] perf [Электронный ресурс]: Linux manual page. — URL: https://man7.org/linux/man-pages/man1/perf.1.html (online; accessed: 2023-05-27).

[19] shm_open(3) [Электронный ресурс]: Linux manual page. — URL: https://man7.org/linux/man-pages/man3/shm_open.3.html (online; accessed: 2023-01-03).

[20] Вызов функций Go из других языков [Электронный ресурс]: Habr. — URL: https://habr.com/ru/companies/vk/articles/324250/ (online; accessed: 2023-05-27).

[21] Примеры плагинов [Электронный ресурс]: GitHub. — URL: https://github.com/semtagg/pacemaker-simple-agent (online; accessed: 2023-01-03).

[22] Системы динамической загрузки библиотек [Электронный ресурс]: GitHub. — URL: https://github.com/semtagg/pacemaker/pulls (online; accessed: 2023-05-27).

[23] Событийно-ориентированная архитектура [Электронный ресурс]: Википедия. Свободная энциклопедия. — URL: https://en.wikipedia.org/wiki/Event-driven_architecture (online; accessed: 2023-01-03).

Приложение А. Flame-графики

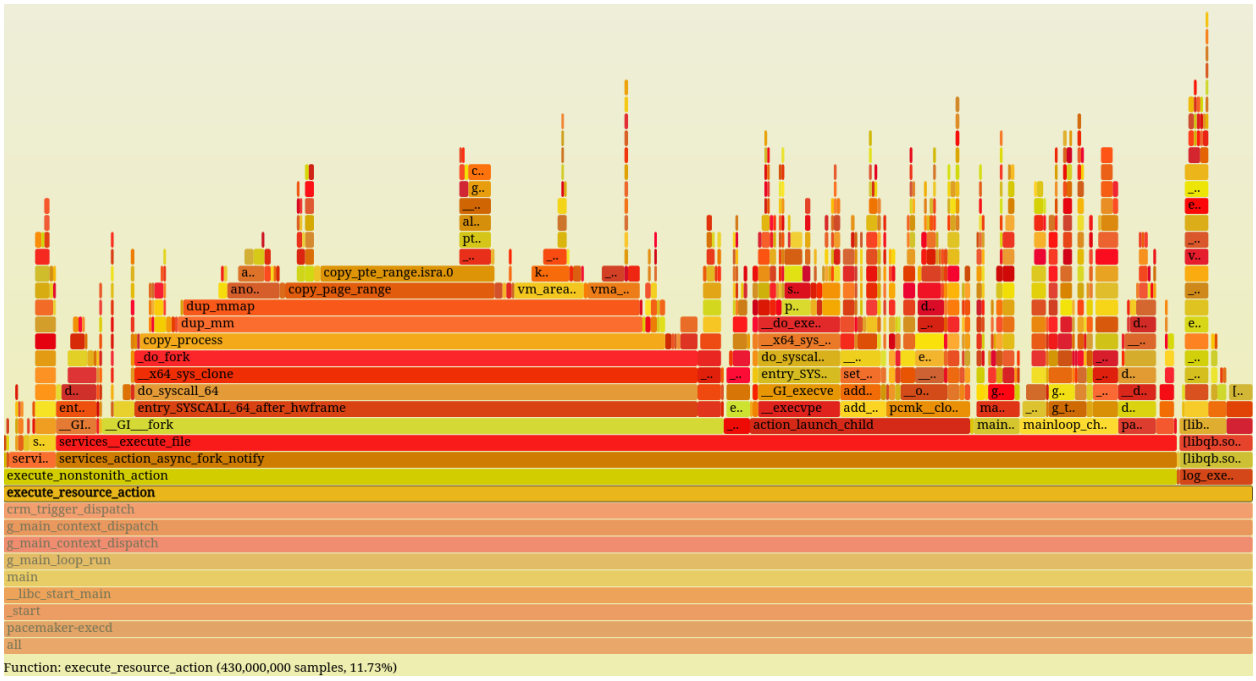

Рис. 4: Flame-график измерений классического pcmk

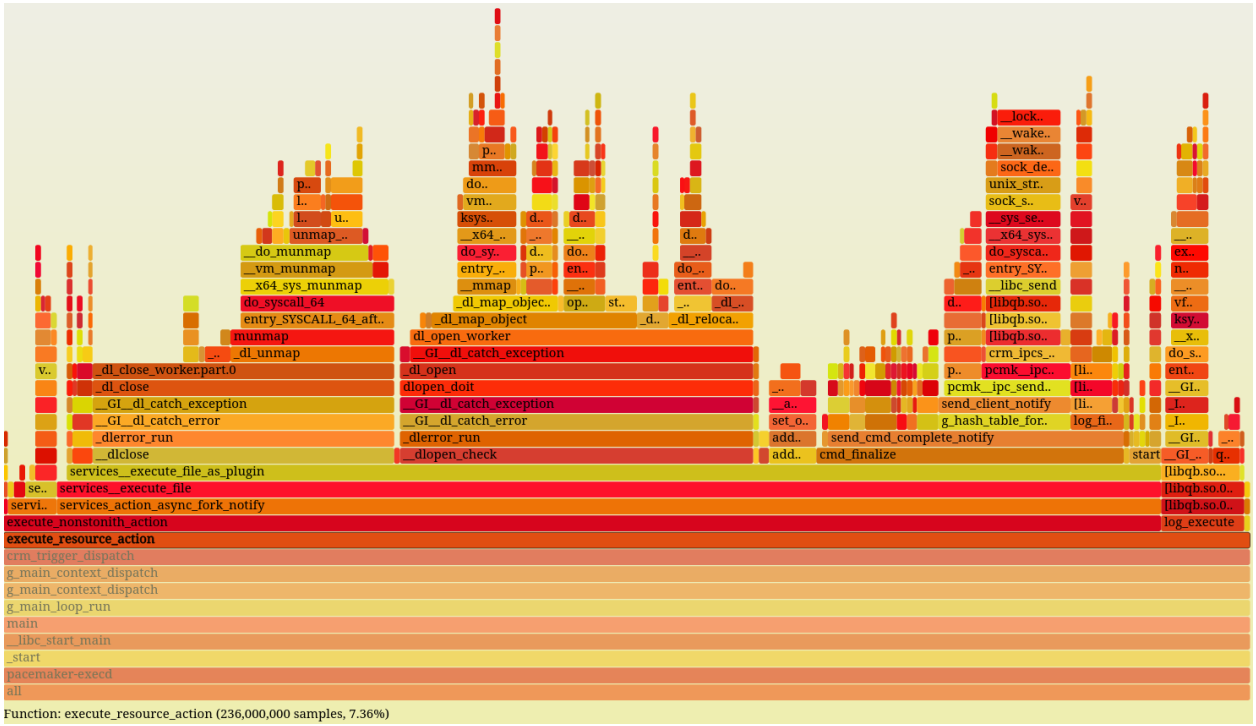

Рис. 5: Flame-график измерений осеннего pcmk

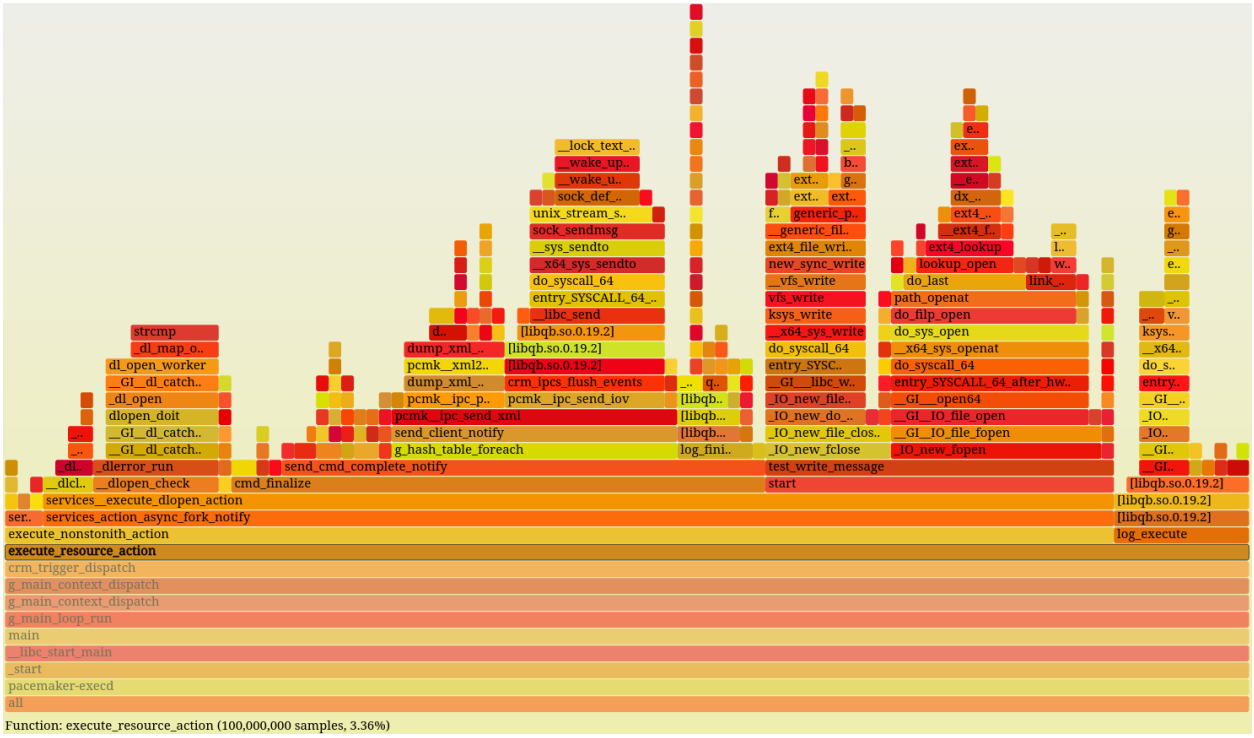

Рис. 6: Flame-график измерений весеннего pcmk с агентами на Си

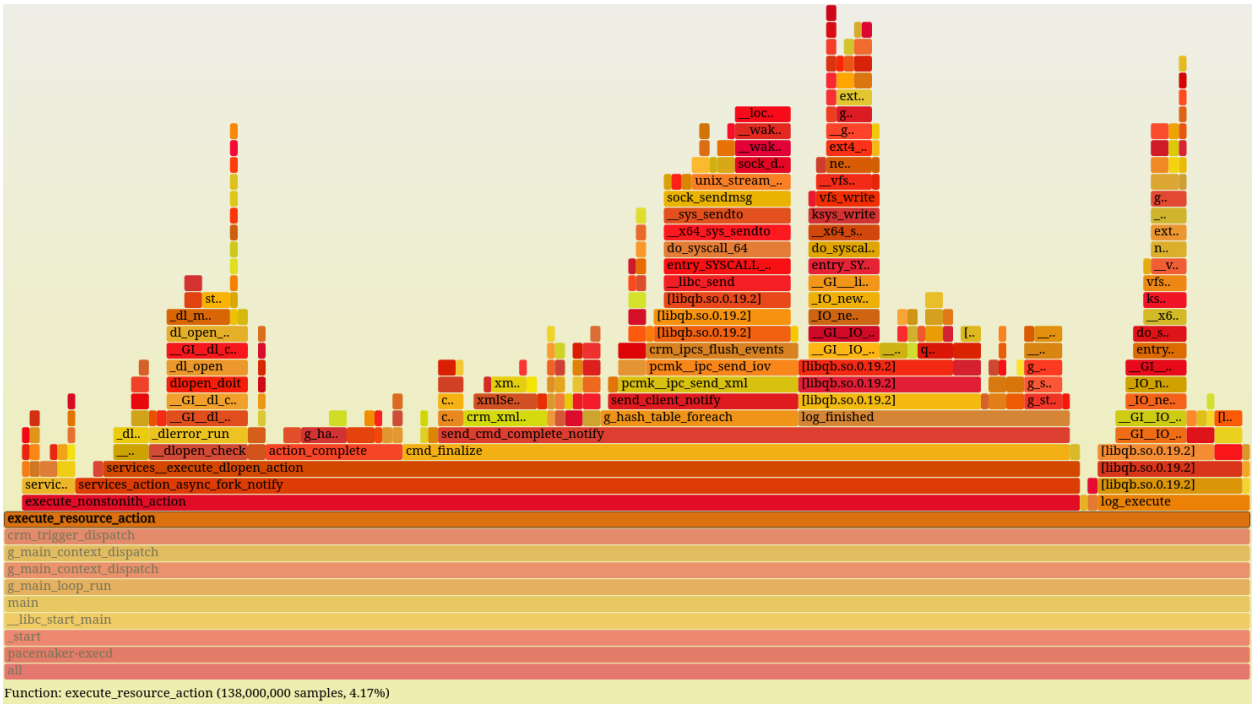

Рис. 7: Flame-график измерений весеннего pcmk с агентами на Go