Санкт-Петербургский государственный университет

Кафедра Системного Программирования

Группа 20.Б11-мм

Pacemaker: оптимизация построения графа

Азарников Иван Евгеньевич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
старший преподаватель кафедры ИАС, К. К. Смирнов

Консультант:
младший инженер-исследователь ООО «КНС ГРУПП» В. А. Кутуев

Санкт-Петербург
2023

Оглавление

Введение

Кластер — связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений, и представляющихся пользователю единой системой. Кластеры бывают разные: вычислительные, с балансировкой нагрузки, высокой доступности. В этой работе рассматриваются кластеры высокой доступности (high-availability clusters). Они обеспечивают восстановление работоспособности ресурсов при аппаратном сбое.

Pacemaker — менеджер ресурсов кластера высокой доступности, поддерживаемый ClusterLabs. Он предназначен для сохранения целостности и уменьшения времени простоя желаемых ресурсов. Также он позволяет вводить зависимости между ними. Например, на одном узле должен запуститься ресурс WebServer, которому необходим запущенный на этом же узле ресурс IPAddr. Pacemaker позволяет задавать порядок запуска ресурсов и предпочтительные узлы для них. Он по-разному обеспечивает доступность ресурсов: перезапуск недоступного узла или миграция на другой.

Отказоустойчивые кластеры широко используются для поддержания важных баз данных, хранения файлов в сети, бизнес-приложений и систем обслуживания клиентов. Pacemaker используется для управления десятком ресурсов. В компании YADRO он используется для управления на порядки большим количеством служб, что выражается в его долгой работе.

Pacemaker реагирует на различные события в кластере (отказ узла, отказ ресурса, запрос администратора) и переводит его в различные состояния. Для перехода в нужное состояние строится граф перехода (transition graph). Он представляет собой список действий, выполнение которых переведет кластер в нужное состояние. Его создание занимает много времени из-за сложной реализации и большого количества изменений. Изменение одного действия может привести к изменению других (запускается «лавинный эффект»).

Чтобы избежать перевычислений при изменении состояния неболь-

шого числа ресурсов было предложено использовать инкрементальность. Однако переход на инкрементальный подход к обработке произошедших в кластере событий может быть сложен, так как необходимо значительно изменить алгоритм обхода графа действий (action graph), сохранив получаемый результат. Поэтому было принято решение использовать DDlog — язык программирования для инкрементальных вычислений. Декларативность DDlog может позволить описать правила построения графа перехода в более ёмком виде.

1. Постановка задачи

Целью работы является переработка алгоритма построения графа для достижения краткости описания и приемлемой производительности с помощью языка DDlog. Для её выполнения были поставлены следующие задачи:

1. Развернуть тестовый пример от компании YADRO и изучить построенный flame-график.
2. Изучить существующий алгоритм построения графа.
3. Реализовать модуль, применяющий ограничения Pacemaker с помощью DDlog:
   - размещение (location constraint),
   - совместное размещение (colocation constraint),
   - порядок (ordering constraint).
4. Протестировать модуль.

2. Обзор

В данном разделе представлен обзор Pacemaker[4] и DDlog[2].

2.1. Pacemaker

Работа Pacemaker поддерживается семью демонами. Они и их зависимости представлены на рис. 1. Наибольший интерес для этой работы представляет планировщик, так как именно он создает граф перехода (transition graph).

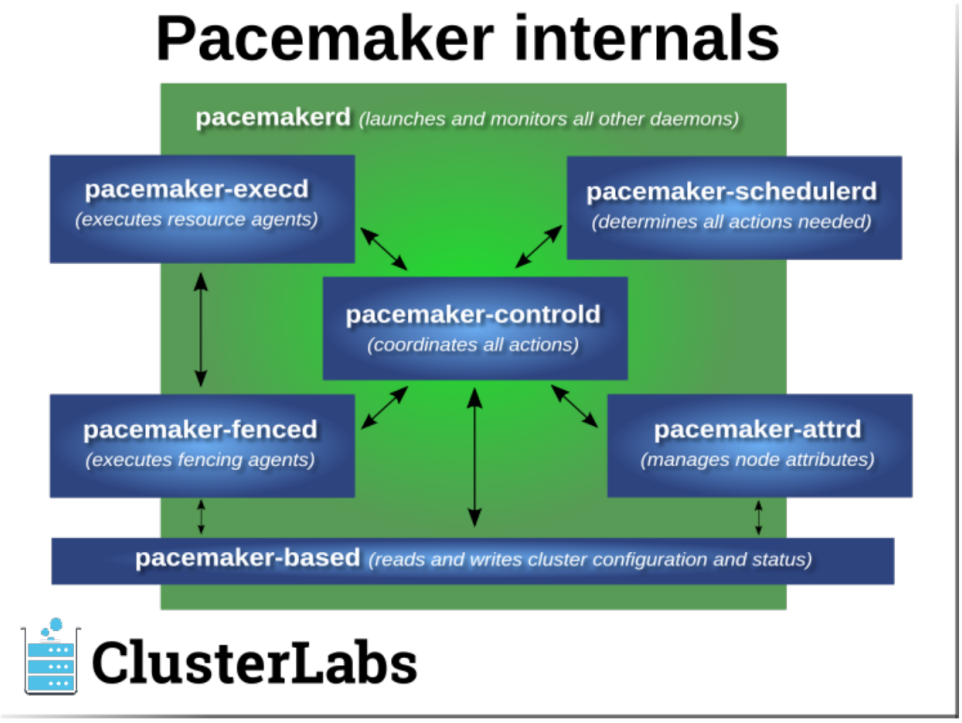

Рис. 1: Pacemaker daemons

2.1.1. CIB

Для поддержания работы кластера Pacemaker хранит всю конфигурацию в CIB (Cluster Information Base). В ней также хранится информация об узлах, ресурсах, ограничениях и статусе кластера. Пример CIB представлен в листинге 1.

Секция configuration содержит информацию об управляемых кластером ресурсах и об их зависимостях. В секции status хранится история ресурсов со всех узлов. Более подробно CIB описана в документации[3].

Листинг 1: CIB

```
<cib crm_feature_set="3.6.0" validate-with="pacemaker-3.5"
    epoch="1" num_updates="0" admin_epoch="0">
    <configuration>
        <crm_config/>
        <nodes>
            <node id="101" uname="pcmk-1"/>
        </nodes>
        <resources>
            <primitive
              id="Email"
              class="service"
              type="exim"/>
        </resources>
        <constraints/>
    </configuration>
    <status/>
</cib>
```

2.1.2. Ресурсы

Ресурс — это сервис, управляемый Pacemaker. Самый простой тип ресурса — примитив. У каждого примитива есть агент (resource agent), предоставляющий стандартизованный интерфейс для управления сервисом.

Типы ресурсов, поддерживаемые Pacemaker:

- примитив;
- группа:

  - состоит из набора различных ресурсов
  - ресурсы в группе запускаются и останавливаются в заданном порядке,
  - ресурсы в группе работают на одном узле;
- клоны:
  - ресурсы, копии которых могут работать на всех узлах одновременно,
  - в основном по одному клону на работающий узел кластера;
- бандл:
  - запуск ресурса в контейнере.

2.1.3. Ограничения

Ограничения в Pacemaker бывают трех видов:

1. размещение (location),
2. совместное размещение (colocation),
3. порядок (ordering).

Location constraint позволяет задать более предпочтительные узлы для ресурсов

```
<rsc_location
    id="loc-1"
    rsc="Webserver"
    node="sles-1"
    score="200"/>
```

Colocation constraint позволяет запустить ресурс вместе с определенным ресурсом на одном узле.

```
<rsc_colocation
    id="coloc-1"
    rsc="Webserver"
    with-rsc="IP"/>
```

Ordering constraint задает порядок, в котором должны запускаться ресурсы.

```
<rsc_order id="order-1" first="IP" then="Webserver"/>
<rsc_order id="order-2" first="Database" then="Webserver"/>
```

2.1.4. Планировщик

При возникновении каких то событий планировщик генерирует граф перехода, с помощью которого Pacemaker переведет кластер в желаемое состояние. На вход планировщик принимает CIB.
Сложности работы с планировщиком.

- Из-за большого количества кода маленькие изменения (в коде) меняют логику неочевидным образом.
- Документация для разработчиков содержит недостаточно информации о правилах построения графа перехода.
- Основная структура данных меняется на разных этапах планирования вплоть до создания графа перехода.

Более подробно scheduler описан в Pacemaker Development

2.2. DDlog

В листинге 2 представлена программа на DDlog, вычисляющая транзитивное замыкание графа. В 1 строке определяется входное отношение. В него клиент отправляет записи. Во 2 строке — выходное отношение, которое вычисляется DDlog'ом автоматически с помощью правил, заданных в 3 и 4 строках.

Листинг 2: Транзитивное замыкание

```
input relation Edge(x: Node_t, y: Node_t)
relation TC(x: Node_t, y: Node_t)
TC(x, y) :- Edge(x, y).
TC(x, y) :- TC(x, z), Edge(z, y).

// from cli
start;
insert Edge(1, 2);
insert Edge(2, 3);
commit;
dump TC;

// output
// TC{.x = 1, .y = 2}
// TC{.x = 2, .y = 3}
// TC{.x = 1, .y = 3}
```

С 6 по 10 строку приведен пример взаимодействия пользователя с объектами DDlog. Любые операции, производимые в клиентской части, должны быть заключены между start и commit. Пользователь вставляет два ребра и распечатывает содержимое отношения TC. Там три записи. Программа отработала, как и ожидалось.

2.2.1. Интеграция с другими языками

Программы, написанные на DDlog, можно интегрировать в программы на C, Rust, Java.

После компиляции и сборки DDlog программы создается статическая библиотека, с которой можно скомпилировать с программой, написанной на C. Это удобно тем, что эту библиотеку можно хранить внутри проекта. В листинге 3 представлен пример того, как можно взаимодействовать с объектами DDlog через API на языке C.

Листинг 3: Интеграция с C

```
#include "ddlog.h"

ddlog_prog *prog = ddlog_run(1,true,NULL,NULL);
table_id EdgesTableID = dlog_get_table_id(prog, "Edge");
ddlog_record **struct_args = (ddlog_record**)
    malloc(2*sizeof(ddlog_record*));

struct_args[0] = src;
struct_args[1] = dst;
ddlog_record *new_record =
    ddlog_struct("Edge", struct_args, 2);

ddlog_transaction_start(prog);
ddlog_cmd *cmd = ddlog_insert_cmd(EdgesTableID, new_record);
ddlog_apply_updates(prog, cmd, 1);
ddlog_transaction_commit(prog);
```

3. Алгоритм построения графа

3.1. Flame-график

Flame-график[1] — визуализация иерархических данных, которая создается для отображения трассировок стека профилируемого программного обеспечения. Flame-график сортирует и агрегирует трассировки на каждом уровне стека, таким образом их количество отражает процент от общего времени, затраченного в этой части кода. Порядок слева направо не имеет принципиального значения, часто это просто сортировка по алфавиту. То же самое и с цветами. Значение имеют только относительная ширина и глубина стека.

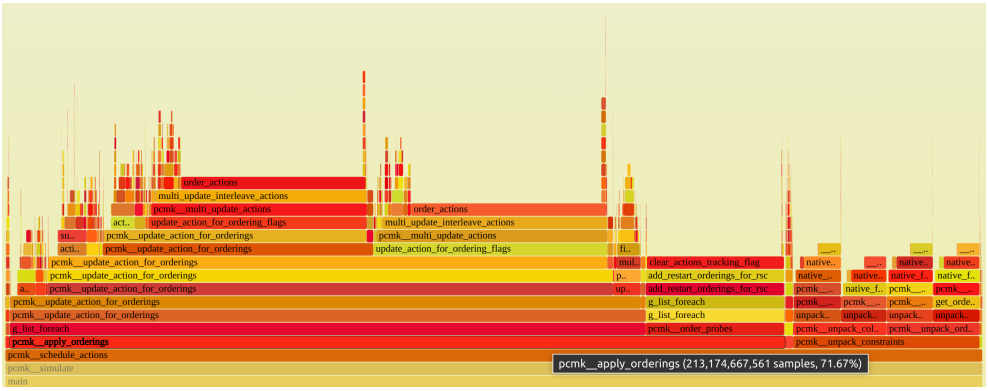

Рис. 2: Flame-график

На рис. 2 представлен flame-график, полученный в результате симуляции работы кластера на основе тестовых данных (46000 узлов в графе действий).

На графике видно, что большую часть времени симуляции заняло применение порядковых ограничений (pcmk__apply_orderings). Перед созданием графа перехода, происходит обновление вершин графа действий — функция pcmk__update_action_for_orderings. Если текущее действие было обновлено, то необходимо обновить и все действия, которые от него зависят. Таким образом эта функция ищет некоторую неподвижную точку графа действий (такое состояние, в котором не нужно обновлять граф). Также стоит обратить внимание на функцию

[1] https://www.brendangregg.com/flamegraphs.html

order_actions. Судя по графику, она часто вызывалась на глубоких уровнях рекурсии.

3.2. Алгоритм

Планировщик отвечает за генерацию графа перехода, для создания которого используется граф действий. В его вершинах — действия, а направленные ребра задают порядок между ними. Граф перехода является подграфом графа действий. Формат графа перехода представлен в листинге 4, пример графа действий на рис 3.

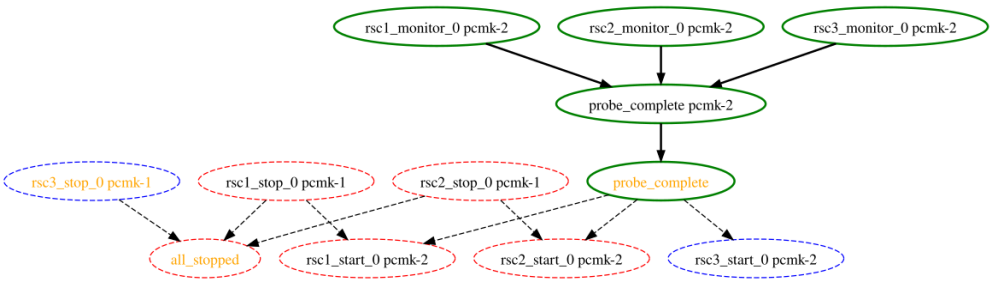

Рис. 3: Пример графа действий

Алгоритм построения графа перехода:

1. распаковать CIB в структуру;
2. создать внутренние ограничения[2];
3. назначить ресурсам узлы (location, colocation);
4. запланировать действия над ресурсами,
5. применить порядковое ограничение (apply orderings),
6. создать граф перехода.

[2]напр. порядок запуска группы

Листинг 4: Граф перехода

```
<transition_graph>
    <synapse>
        <action_set>
        </action_set>
        <input>
            <trigger/>
            ...
            <trigger/>
        </input>
    </synapse>
    ...
</transition_graph>
```

Исходя из данных на рис. 2, 5-ый шаг алгоритма занимает много времени из-за рекурсивной функции pcmk__update_action_for_orderings, которая в свою очередь тратит много времени на выполнение функции order_actions. Однако order_actions исполняется за $O(1)$, значит она вызывается очень часто.

4. Реализация

4.1. Ограничения размещений

Реляция — основная сущность, которой оперирует DDlog. Поток данных поступает во входные реляции (input relation), проходит через промежуточные (relation) и оказывается в выходных реляциях (output relation) при помощи правил.
В Листинге 5 представлена реализация применения ограничений размещения на DDlog.

Листинг 5: Вычисление ограничений размещения

```
typedef id_t = u64

input relation Node (id: id_t)
input relation Rsc(
    id: id_t,
    variant: rsc_variant_t,
    children: Vec<id_t>
)
input relation LocationConstraint (
    id: id_t,
    rsc: id_t,
    node: id_t,
    score: u64
)
input relation ColocationConstraint (
    id: id_t,
    rsc: id_t,
    with_rsc: id_t
)
relation Location (rsc: id_t, node: id_t)
relation Colocation (rsc: id_t, with_rsc: id_t, node: id_t)
output relation TotalLocation (rsc: id_t, node: id_t)
```

Входные данные поступают в реляции Node, Rsc, LocationConstraint и ColocationConstraint. После чего преобразуются, в соответствии с правилами, и поступают в Location и Colocation. Они вычисляются при помощи входных реляций и содержат промежуточные вычисления. Реляция TotalLocation содержит обобщенную информацию о размещении ресурсов. Она вычисляется при помощи Location и Colocation.

4.2. Порядковые ограничения

Порядковые ограничения задают порядок запуск ресурсов (остановка в обратном порядке). Таким образом в графе действий появляются ребра между действиями типа start и stop над различными ресурсами.

Рассмотрим возможные комбинации порядка запуска ресурсов:

1. примитив → примитив;
2. примитив → клон;
3. клон → примитив;
4. клон → клон.

Вариации с группами не рассматриваются, так как группы — это синтаксический сахар. Они раскрываются в ограничения размещения (location, colocation) и порядка (ordering). Бандл не рассматривается из-за того, что логика обновления действий над бандлами схожа с логикой обновления действий над клонами.

У клонов есть экземпляры на разных узлах, при применении порядка это нужно учитывать. Если клон стоит первый, то второй ресурс должен запуститься после того, как все экземпляры клона были запущены.

Обработка комбинаций 1 и 2 проста и не занимает много времени. При обработке комбинаций 3 и 4 в граф могут добавиться новые ребра, что повлечет за собой обновление зависимых действий (увеличение глубины рекурсии). Ребра добавляются в граф действий в функции

order_actions, которая занимает немалую долю времени исполнения на графике 2, поэтому было принято решение выразить логику обновления ребер графа (поиск дельты).

Листинг 6: Вычисление порядковых ограничений

```
input relation &Action(
    id: id_t,
    rsc_id: Option<id_t>,
    task: task_t,
    interval: uint_t,
    node_id: Option<id_t>,
    flags: flag_t,
    deps_num: uint_t,
)
input relation ActionsGraph(
    first_id: id_t,
    then_id: id_t,
    order_type: flag_t
)
relation ClonePrimitiveStartActions(
    clone: Ref<Action>,
    child: Ref<Action>
)
relation CloneCloneStartActions(
    first: Ref<Action>,
    then: Ref<Action>
)
output relation EdgesToDelete(from: id_t, to: id_t)
output relation EdgesToInsert(from: id_t, to: id_t)
```

В листинге 6 представлены основные реляции необходимые для обновления графа действий. В реляции Action хранится необходимая информация о действиях, ActionsGraph представляет направленный граф действий с флагами. ClonePrimitiveStartActions и CloneCloneStartActions

хранят информацию о порядковых ограничениях для комбинаций 3 и 4. EdgesToDelete и EdgesToInsert — дельта графа действий (то, что отправляется pacemaker'y).

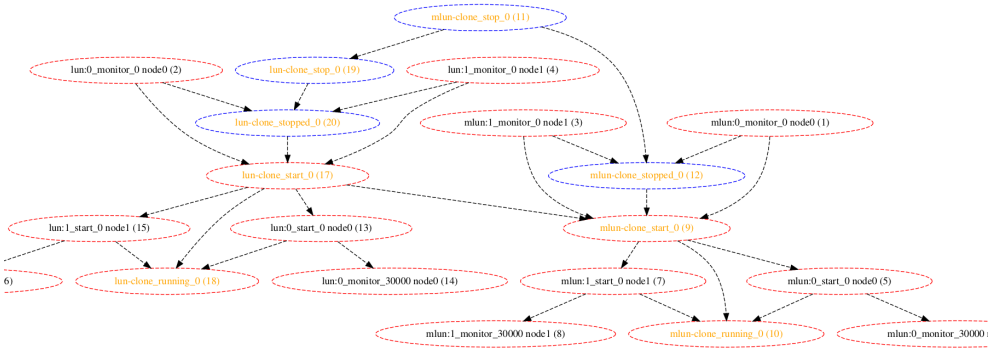

Рис. 4: Граф действий до обновлений

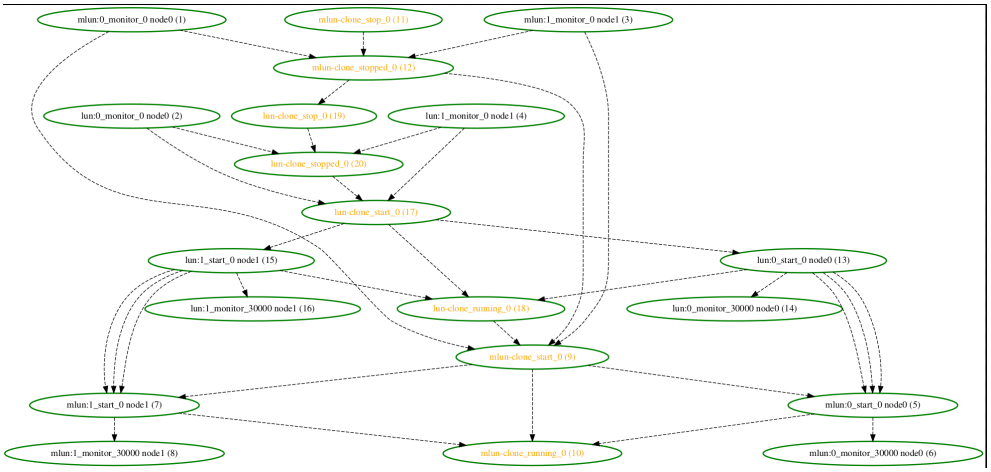

Рис. 5: Граф действий после обновлений

5. Тестирование

Для проверки работоспособности кода была написана конфигурация[3] кластера с двумя узлами и двумя клонами.

На рис. 4 представлено состояние графа действий до обновления, на рис. 5 — после обновления. Вывод программы на DDlog представлен в листинге 7. Это и есть искомая дельта графа.

Листинг 7: Вывод

```
EdgesToDelete:
EdgesToDelete{.from = 11, .to = 19}: +1
EdgesToDelete{.from = 17, .to = 9}: +1

EdgesToInsert:
EdgesToInsert{.from = 12, .to = 19}: +1
EdgesToInsert{.from = 13, .to = 5}: +1
EdgesToInsert{.from = 15, .to = 7}: +1
EdgesToInsert{.from = 18, .to = 9}: +1
```

[3] https://github.com/esvault/ddlog_immersion/blob/master/pcmkr/sched_io/clones-2.xml

6. Заключение

Результаты выполнения работы следующие:

- развернут тестовый пример и изучен flame-график,
- изучен алгоритм построения графа,
- реализован модуль, применяющий ограничения Pacemaker с помощью DDlog,
- протестирован реализованный модуль

Исходный код реализации расположен на GitHub [1].

Список литературы

[1] DDlog immersion GitHub. — URL: https://github.com/esvault/ddlog_immersion/tree/master/pcmkr (дата обращения: 8 ноября 2023 г.).

[2] Differential Datalog GitHub. — URL: https://github.com/vmware/differential-datalog (дата обращения: 8 ноября 2023 г.).

[3] Pacemaker Explained. — URL: https://www.clusterlabs.org/pacemaker/doc/2.1/Pacemaker_Explained/pdf/Pacemaker_Explained.pdf (дата обращения: 8 ноября 2023 г.).

[4] Pacemaker GitHub. — URL: https://github.com/ClusterLabs/pacemaker (дата обращения: 8 ноября 2023 г.).