ОБ ОДНОМ ПОДХОДЕ К МОДЕЛИРОВАНИЮ СУПЕРКОМПЬЮТЕРНЫХ КОМПЛЕКСОВ

П.А. Швец, В.В. Воеводин, С.И. Соболев

Научно-исследовательский вычислительный центр Московского государственного университета имени М. В. Ломоносова (НИВЦ МГУ)

Введение

Поддержка автономного функционирования суперкомпьютерного центра — одна из важнейших задач, с которой сталкиваются их владельцы и администраторы. Анализ деятельности суперкомьютерных центров как в России, так и за рубежом показал, что каждый решает эту задачу своими силами, создавая индивидуальные и непереносимые комплексы. Этот же вопрос и встал перед нами в рамках работ по обеспечению эффективной работы Суперкомпьютерного комплекса МГУ.

Нами был предложен [1] метод контроля на основе модели суперкомпьютерного комплекса, представленного в виде расширенного мультиграфа. Вершины графа описывают физические (ЦПУ, кондиционер) и логические (область подкачки, файловая система) компоненты суперкомпьютерного комплекса. Рёбра графа описывают связи между компонентами: например, вершина «стойка» может быть связана с вершиной «шасси» связью «содержит», вершина «кондиционер» - связана с вершиной «горячий коридор» связью «охлаждает». Пример модели показан на рисунке 1.

С вершинами и связями ассоциируются атрибуты, описывающие состояние компонент (температура, IP-адрес), правила, позволяющие преобразовывать атрибуты (например, вычислять скорость) и реакции, срабатывающие, когда атрибуты принимают определённое значение (например, информирование системного администратора). Далее система контроля, основываясь на описанной модели, производит получение, обработку и анализ реальных данных с различных аппаратных и программных сенсоров, а с помощью набора правил и реакций происходит контролирование штатной работы комплекса.

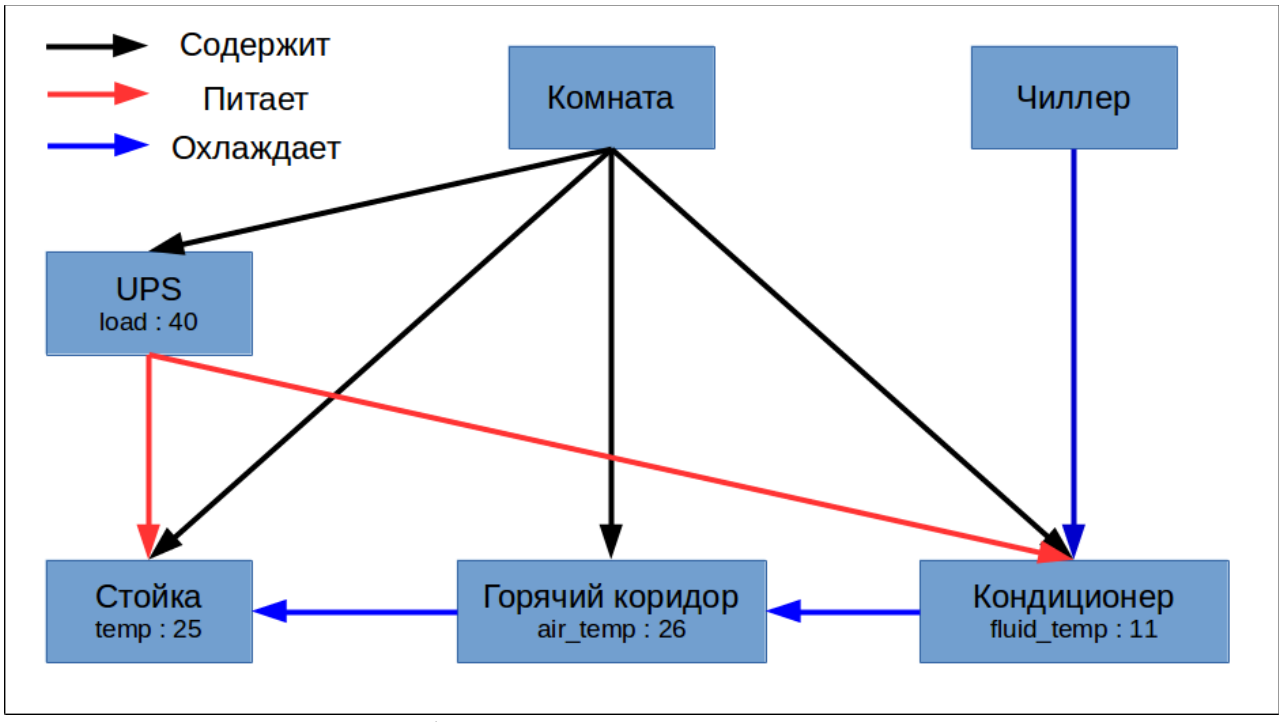

Рис. 1. Пример части модели суперкомпьютера

Данный подход позволяет обрабатывать ряд ситуаций, которые сложно отслеживать в рамках раздельного мониторинга всех компонент, при котором не учитываются физические и логические связи между разными компонентами.

Например, при проблемах с одним источником питания можно отключать только то оборудование, которое питается непосредственно от него, или следить за количеством узлов в конкретной очереди с ошибками определённого типа.

Также наличие модели со связями полезно для систематизации знаний о суперкомпьютерных комплексах и потенциальных источниках сбоев. Каждый объект рассматривается как потенциально сбойный, и проверятся, что с ним может произойти, и как это повлияет на соседние компоненты. Подробней принципы моделирования описаны в статье [1], а настоящая работа посвящена детальному описанию реализованного нами подхода к моделированию суперкомпьютерных комплексов.

Анализ средств для работы с графами

На основе изучения характеристик современных суперкомпьютерных комплексов стало понятно, что для представления модели СК в виде графа необходимо, чтобы система работы с графами удовлетворяла следующим требованиям:

- Быстрая работа с большими графами. Сотни тысяч вершин, миллионы атрибутов - таков, по нашей оценке, масштаб требуемых графов для описания моделей суперкомпьютеров с верхних строчек списка Top500.
- Поддержка кратных рёбер. По факту в модели некоторые объекты могут быть связаны двумя и большим типом связей. Например вершина «стойка» должна быть связана с вершиной «шасси» связями «содержит» и «питает».
- Поддержка атрибутов разных типов(строки, целые числа, числа с плавающей запятой) для вершин и рёбер.

После этого были изучены существующие средства для работы с большими графами. Мы рассмотрели такие инструменты, как Gephi[2] и Tulip/porgy[3].Gephi больше ориентирован на интерактивную работу пользователя с графами, а Tulip/porgy, имеющий средства для автоматического преобразования графов, использовал неподходящую нам систему правил преобразований. Доработка этих инструментов была признана нецелесообразной и было принято решение об исследовании более низкоуровневых средств.

Анализ низкоуровневых библиотек и программных средств для работы с графами

Существует огромное множество библиотек на самых разных языках программирования, поэтому мы рассмотрели наиболее популярные из них и выбрали удовлетворяющую нашим требованиям.

- Boost Graph Library [4] (BGL) — популярная библиотека, отвергнутая нами из-за её основного языка программирования - C++. Требования к разрабатываемой системе дорабатывались во время разработки, что могло бы потребовать очень больших временных затрат при использовании C++ как основного языка проекта.
- Graph-tool [5] – интерфейс для BGL на языке программирования Python. Скорее всего производительности языка не хватило бы для обработки требуемых нам масштабов данных.
- Neo4j[6] – графовая база данных, написанная на языке Java и ориентированная на работу с высоконагруженными сервисами.

Описание Neo4j

В итоге для разработки был выбран язык Java, к достоинствам которого можно отнести быструю разработку и наличие множества готовых библиотек, и база данных Neo4j, как средство для работы с графами. Neo4j – это нереляционная база данных, оперирующая понятиями узлы (nodes), связи (relationships) и атрибуты (properties), которые могут устанавливаться как на узлы, так и на связи. Это полностью подходит под наши требования к описанию модели, и вдобавок neo4j предоставляют поддержку следующих полезных особенностей:

- Механизм автоматическое кеширования позволяет хранить часто используемые данные в памяти, редко используемые — на диске. Данная особенность позволяет эффективно работать с графами очень больших размеров, что может понадобиться для описания суперкомпьютеров экзафлопсного масштаба.
- Быстрый поиск по свойствам (индексирование) напрямую влияет на скорость исполнения запросов к системе, а значит и на максимальное количество обрабатываемых запросов в секунду, позволяя обрабатывать данные от большего числа сенсоров с большей частотой.
- ACID-транзакции [7] — данная модель транзакций облегчает техническую обработку параллельных запросов.
- Готовый механизм сохранения/загрузки с диска позволяет продолжать работу системы после остановки без потери прошлых данных.

Интерфейс работы с графами (API)

Мы не стали закладываться на работу только с Neo4j и реализовали свой промежуточный интерфейс для работы с графами, используя Neo4j как одну из возможных реализаций. Если в какой-то момент нам понадобится функционал, отсутствующий в Neo4j, но присутствующий в какой-то другой системе — мы сможем перейти на неё без существенного изменения логики работы системы. Исходя из наших требований, API для работы с графом включает в себя следующие возможности:

- Создать объект.
- Создать связи между созданными объектами.
- Установить и получить атрибуты по объекту или связи.
- Набор функций для удаления заданных объектов, связей и атрибутов.

API для поиска по графу содержит следующие возможности:

- Найти все объекты/связи, у которых есть атрибут с заданным именем.
- Найти все объекты/связи, у которых есть атрибут с заданным именем и значением.
- Найти все объекты/связи, у которых есть строковые атрибуты с заданным именем и значением, подходящим под заданный шаблон.

На основе этих двух API мы реализовали все внутренние сервисы, позволяющие сохранять в модели более сложные объекты, обеспечивающие возможность обновления атрибутов, правил, вызов реакций, и реализующие внешний протокол запросов к графу.

Язык описания модели

После разработки системы встал вопрос об описании самой модели. Наиболее простой метод — описание модели на языке Java с помощью разработанного API. К сожалению, этот подход имеет существенный недостаток: Java — довольно «многословный» язык, а для описания модели хватило бы лишь небольшого подмножества языка. Также сама необходимость знать и использовать для описания язык Java может восприниматься неоднозначно.

Была предпринята попытка разработки собственного языка описания модели, который бы потом транслировался в Java. Первая версия преобразовывала код построчно с помощью набора регулярных выражений, но стало понятно что ни развивать, ни поддерживать такой вариант невозможно: требовались все более сложные языковые конструкции, а их реализация с помощью регулярных выражения получалась очень сложной и громоздкой.

Дальнейшим шагом мы изучили готовый средства, генерирующие трансляторы по формальной грамматике языка (например ANTLR [8]), но после исследования наработок по этому направлению решили, что полноценная реализация собственного описательного языка займёт слишком много времени.

В итоге, в качестве основного языка описания моделей был выбран язык Python [9] и использован специальный интерпретатор Jython [10], исполняющий код на JVM (виртуальная машина, на которой исполняются Java программы) и позволяющий без каких-либо дополнительных усилий использовать Java-классы из программы, написанной на Python. Сам по себе Python — очень простой и понятный язык, описать на нём модель сможет даже человек, не знакомый с ним, руководствуясь нашими примерами и документацией. Мы разработали модуль, предоставляющий простой интерфейс для исходного API на языке Java.

Первая часть описания модели — задание атрибутов, правил и реакций.

- Описание атрибутов совпадает с заданием словаря на языке Python в формате «имя : значение по-умолчанию». Значение по-умолчанию используется для задания типа переменной. Значения атрибутов могут быть целыми и вещественными числами, строками и булевыми константами True/False.

```
attributes = {
    "name1" : "value"
    "name2" : 10,
    "name3" : 4.0,
    "name4" : True
}
```

- Правила описываются в стиле словаря, в формате «имя : конструктор правила». Имя обозначает имя атрибута, который будет создаваться правилом. Конструкторы правил индивидуальны в каждом случае и полностью описаны в документации [14].

```
rules = {
    "rule1" : RuleClass1(param1, param2, ...),
    "rule2" : RuleClass2(param1, param2, ...),
}
```

- Реакции описываются в стиле словаря, в формате «(имя, значение) : конструктор реакции». Реакция срабатывает, когда соответствующий атрибут принимает указанное значение. В качестве реакции мы создали 4 стандартных типа (Info, Warning, Danger, Critical), но у пользователя есть возможность модифицировать реакции, добавляя вызов произвольных скриптов или создавая собственные реакции.

```
reactions = {
    ("name1", "value") : Reacton(Warning("message")),
    ("name2", 30) : Reacton(Danger("message"))
}
```

Вторая часть описания модели — создание объектов и связей.

- Для создания объектов используется функция CreateObjects(attributes, rules, reactions, count) и CreateObject(attributes, rules, reactions). Параметрами данных функций являются описанные выше атрибуты правила и реакции, а также количество объектов, которые необходимо создать. Есть возможность указывать не один словарь в каждый параметр, а список из словарей, описывая необходимые атрибуты в самом вызове функции. Первая функция возвращает список объектов, а вторая — один объект.

```
objects = CreateObjects([attributes, { "type" : "value"  }]
    , rules, reactions, 10)

obj = CreateObject([attributes, { "type" : "value"  }]
    , rules, reactions)
```

- Для создания связей между объектами используется набор функций OneToOne, OneToEvery, AllToAll и др. (описание каждой функции доступно в документации). Параметрами этих функций являются сущности, которые надо соединить и список типов связей.

```
OneToEvery(obj, objects, "type1", "type2")
```

```
# атрибуты ИБП
ups_attributes : {
    "load" : 0,
    "state" : "normal"
}

# правило для проверки загрузки ИБП
check_load : {
    "load_ok" : LowerThreshold("load", 10)
}

# реакция на случай низкой загрузки или изменения состояния ИБП
react_load : {
    ("load_ok", false) : Reaction(Warning("UPS load is below 10%"))
    ("state", "bypass") : Reaction(Critical("UPS is in the bypass mode!"))
}

# правило для проверки температуры стойки
check_temp : {
    "temp_ok" : UpperThreshold("temp", 30)
}

# реакция на случай перегрева
react_temp : {
    ("temp_ok", false) : Reaction(Danger("Rack is too hot!"))
}

# создание объектов и задание для них атрибутов, правил и реакций
room = CreateObject()
chiller = CreateObject()
ups = CreateObject(ups_attributes, check_load, react_load)
fan = CreateObject({"fluid_temp" : 0})
hot_aisle = CreateObject({"air_temp" : 0})
rack = CreateObjects({"temp" : 0}, check_temp, react_temp, count = 10)

# создание связей между компонентами
OneToOne(room, ups, "contains")
OneToOne(room, fan, "contains")
OneToOne(room, hot_aisle, "contains")
OneToEvery(room, rack, "contains")

OneToOne(room, chiller, "contains")
OneToOne(room, chiller, "contains")

OneToOne(ups, fan, "power")
OneToEvery(ups, rack, "power")

OneToOne(chiller, fan, "chill")
OneToEvery(fan, rack, "chill")
```

Рис. 2. описание модели, изображённой на рис.1, с помощью разработанного API

Методика построения модели

Суперкомпьютеры состоят из множества различных компонент, что существенно затрудняет построение полной описывающей модели, но чем больше компонент будет внесено в такую модель, тем более полный контроль над комплексом будет у системы мониторинга и реагирования.

На данный момент мы описываем следующие подсистемы суперкомпьютера (в скобках указаны примеры объектов из этой подсистемы):

- Система электропитания (источники бесперебойного питания, модули с батареями).
- Система охлаждения (чиллеры, кондиционеры, мониторинг среды).
- Управляющая часть (узлы доступа и компиляции, очереди задач).

- Вычислительная часть (шасси, узлы, диски, память).
- Файловая система (зависит от типа ФС).
- Ethernet сеть (коммутаторы, порты).
- Infiniband сеть (коммутаторы, менеджер сети).

Модель можно описывать вручную или, пользуясь довольно регулярной структурой некоторых компонент, можно частично генерировать программно. Индивидуальны данные объектов (ip-адрес, серийный номер и т.д.) в таком случае подгружаются из внешних CSV-файлов.

Одним из направлений работы над проектом является разработка инструментария для автоматизированного построения модели. Сгенерированная модель будет требовать дальнейшей ручной доработки, но часть рутинной работы будет уже выполнена.

Визуализация модели

В ходе разработки довольно часто возникало желание визуально оценить созданную модель. Это необходимо как для эмпирической проверки корректности построения, так и может быть использовано в демонстрационных или обучающих целях. Однако полностью отобразить модель практически нереально — алгоритмы расположения графов на плоскости не справляются с графами такого масштаба или показывают совершенно нечитаемый результат.

Но если не ставить задачу визуализации полного графа, а использовать регулярность многих компонент графа, то задача визуализации имеет решение. По многим типам связей модель выглядит как дерево с наличием похожих поддеревьев, которые можно разбивать по уровням и группировать, отображая в итоге метаграф из сгруппированных подграфов.

Был разработан прототип веб-сервиса, который производит «агрегацию» объектов модели и отображает получившийся метаграф. Результат его работы можно видеть на рис. 4 — это визуализация части модели суперкомпьютера «Чебышёв» [11]. Цифры на связях показывают, сколько «похожих» поддеревьев было объединено в одну вершину.

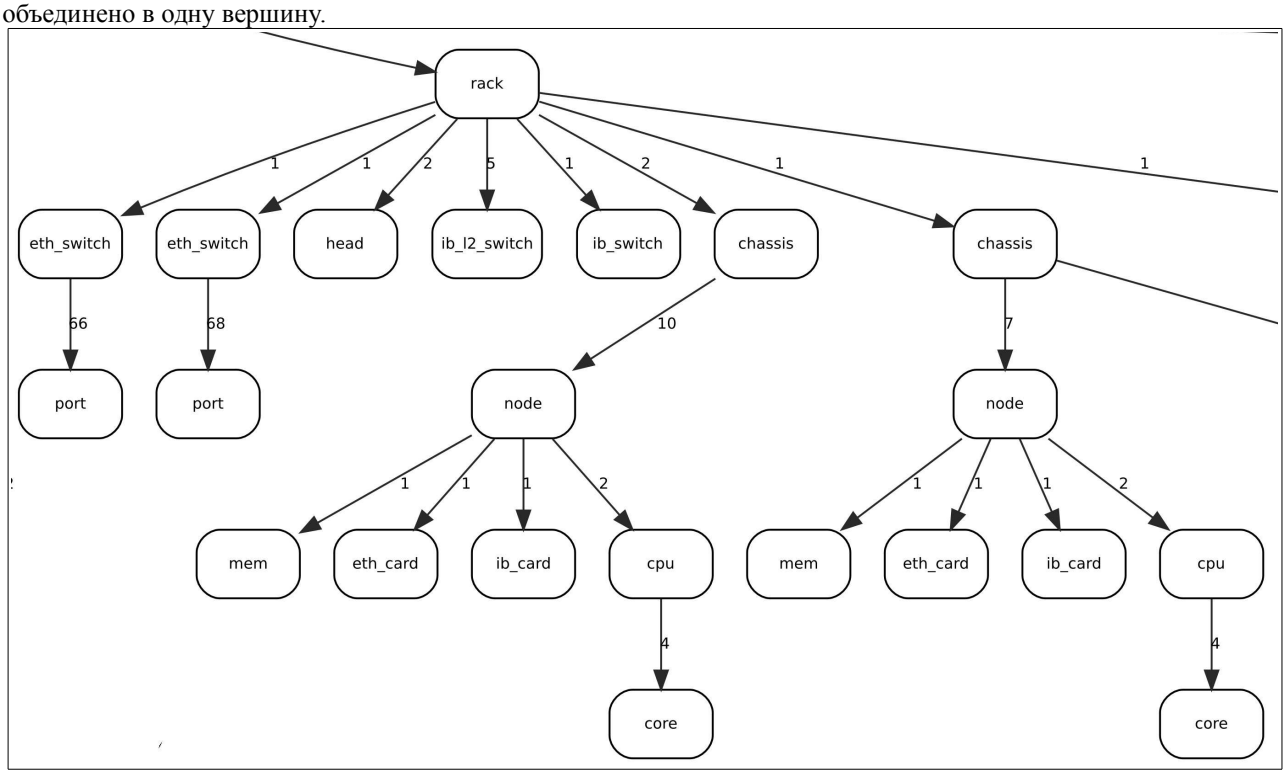

Рис. 3. Пример визуализации «агрегированной» модели

Работа с системой

После описания модели можно произвести запуск самой системы контроля и настроить импорт реальных данных. Управление системой происходит с помощью http-запросов к разработанной компоненте. Данный метод используется и для автоматического импорта данных в модель и для ручного управления. Такой подход позволяет управлять всем мониторингом из обычного браузера и может легко интегрироваться в другие веб-сервисы, например, для визуализации данных.

Наполнение системы реальными данными

Для нужд оперативного контроля информацию от суперкомпьютера можно распределить по трём категориям:

- Информация от инфраструктурного оборудования — охлаждение, питание и пр. Такой информации достаточно мало, но требуется оперативная обработка и реагирование для сохранения оборудования в целостности. Требуемое время оповещения и реагирования - меньше минуты. Также здесь используется механизм «активных событий» - при некоторых событиях данные в модель заносятся сразу при наступлении события, в обход стандартного механизма опроса.
- Служебная информация от основных систем суперкомпьютера — файловая система, система очередей, служебные узлы. Данной информации значительно больше, ошибки могут повлечь проблемы с доступом к суперкомпьютеру, но не могут навредить оборудованию. Требуемое время оповещения и реагирования — до несколько минут.
- Служебная информация с узлов — локальные проблемы на узлах, выход узлов из строя. Ошибки, которые могут проявляться на этом уровне, повлияют лишь на некоторые пользовательские задачи и не могут нанести серьёзный вред оборудованию. Требуемое время оповещения и реагирования — десятки минут, часы.

Поток информации первой и второй категории не превысит нескольких сотен обновлений атрибутов в секунду, а поток информации третей категории не имеет фиксированных требований к частоте съёма, которую можно регулировать для стабильной работы.

На данный момент система на среднем компьютере и графе, помещающемся в память, обрабатывает около 2-3 тысяч обновлений атрибутов в секунду в режиме однопоточной работы с базой данных. Исходя из этих данных, на системах «Чебышёв» и «Ломоносов» суперкомпьютерного комплекса МГУ мы установили следующую частоту съёма:

	«Чебышёв» (~600 узлов)	«Ломоносов» (~5000 узлов)
1-ая категория	Опрос 1 раз в минуту + активные события	Опрос 1 раз в минуту + активные события
2-ая категория	Опрос 1 раз в минуту	Опрос 1 раз в минуту
3-я категория	Опрос 1 раз в минуту	Опрос 1 раз в 10 минут

Заключение

Разработка ядра системы сейчас практически завершена, подход проходит апробацию в Суперкомпьютерном комплексе Московского Государственного Университета. Разработанная система доступна под открытой MIT лицензией [13][14].

В дальнейшем мы планируем развивать средства визуализации и автоматической генерации модели и разработать библиотеку стандартных компонент суперкомпьютера и средств мониторинга для облегчение процесса создания модели.

Работа выполняется при финансовой поддержке РФФИ, грант №12-07-33047.

ЛИТЕРАТУРА:

1. А.С. Антонов, В.В. Воеводин, Вад.В. Воеводин, С.А. Жуматий, Д.А. Никитенко, С.И. Соболев, К.С. Стефанов, П.А. Швец, Разработка принципов построения и реализация прототипа системы обеспечения оперативного контроля и эффективной автономной работы суперкомпьютерных комплексов. Вестник УГАТУ 18, 2 (2014), 227–236.
2. M. Bastian, S. Heymann, M. Jacomy. Gephi: an open source software for exploring and manipulating networks. International AAAI Conference on Weblogs and Social Media 2009.
3. B. Pinaud, G. Melançon, J. Dubois. PORGY: A Visual Graph Rewriting Environment for Complex Systems. Computer Graphics Forum - Eurographics Conference on Visualization (EuroVis 2012) special issue, 31 (3pt4), pp. 1265-1274.
4. http://www.boost.org/doc/libs/1_55_0/libs/graph/doc/index.html
5. http://graph-tool.skewed.de/
6. http://www.neo4j.org/
7. J. Gray. The Transaction Concept: Virtues and Limitations. Proceedings of the 7th International Conference on Very Large Databases: pages 144—154, 1981
8. http://www.antlr.org/
9. www.python.org
10. http://www.jython.org/
11. http://parallel.ru/cluster/skif_msu.html

12. http://parallel.ru/cluster/lomonosov.html
13. Полный исходный код проекта: https://github.com/srcc-msu/octotron_core
14. Рабочее окружение для создания модели на языке Python: https://github.com/srcc-msu/octotron