РАЗДЕЛ I

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ

Децентрализованное управление нагрузкой в распределенной вычислительной среде*

О. С. Естехин

Институт системного анализа Российской академии наук (ИСА РАН)

Централизованное распределение нагрузки

Одним из основных преимуществ распределенной модели вычислений по сравнению с вычислениями на одном компьютере является возможность использования более мощных вычислительных ресурсов, чем имеются в наличии локально. С точки зрения пользователей это позволяет решать задачи за меньшее время, а с точки зрения поставщиков вычислительных ресурсов это позволяет более эффективно использовать вычислительные мощности за счет обслуживания нескольких пользователей. Задача распределения и балансировки нагрузки в общем случае позволяет выяснить, как наиболее эффективно обработать все пользовательские запросы и как равномерно загрузить все имеющиеся вычислительные мощности.

* Работа выполнена при поддержке Президиума РАН (программа фундаментальных исследований 15П, проект 1.4).

При использовании механизма распределения нагрузки обычно получается система, которая обладает следующими характеристиками:

- Балансировка нагрузки — распределение запросов между серверами для предотвращения перегрузки отдельного сервера.
- Устойчивость — обеспечение корректной работы системы в целом даже в случае сбоя отдельного сервера.
- Масштабируемость — увеличение производительности при увеличении количества серверов.

Традиционный подход к распределению нагрузки заключается в создании центрального компонента, обычно называемого балансировщиком нагрузки. Все клиентские запросы поступают только на балансировщик, который определяет, на какой сервер отправить каждый поступивший запрос.

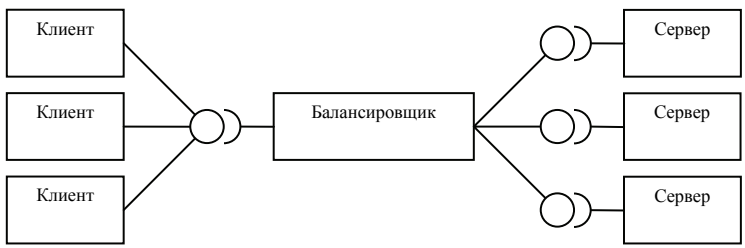

Рис. 1. Использование централизованного балансировщика

К недостаткам централизованного балансировщика относятся:

- Необходимость синхронизации информации о внутреннем состоянии каждого отдельного сервера между самим сервером и балансировщиком. Для корректной работы балансировщика каждый сервер должен уведомлять балансировщик о своем включении и выключении, а так же о текущем уровне загруженности. Расхождение между информацией, имеющейся у балансировщика, и реальным состоянием сервера может приводить как к перегруженности или недогруженности сервера, так и к попыткам использования несуществующего сервера или игнорированию простаивающего сервера.

- При увеличении количества запросов сам балансировщик может стать узким местом, ограничивающим производительность системы в целом. То есть масштабируемость и максимальная производительность системы ограниченны максимальной пропускной способностью балансировщика.
- Критический сбой в работе балансировщика приводит к сбою в работе всей системы. Кроме ошибок в работе самого балансировщика, к остановке системы могут привести так же события, напрямую не связанные с качеством реализации балансировщика, например, потеря связи только между хостом балансировщика и остальной системой приведет к остановке системы даже не смотря на то, что сервера, непосредственно выполняющие работу, останутся доступными.

Децентрализованное распределение нагрузки

В настоящее время децентрализованные системы получают всё большее распространение. Другими названиями этой же концепции являются одноранговая или пиринговая (от английского peer-to-peer) система. Отличительными свойствами децентрализованной системы являются:

- Равноправие участников, которые называются узлами. Все узлы выполняют одинаковые функции, принимая, по необходимости, роли клиента или сервера только при выполнении конкретных запросов.
- Отсутствие центрального управляющего сервера.
- Отсутствие центрального маршрутизатора.

На практике так же существуют гибридные системы, совмещающие некоторые характеристики централизованных и децентрализованных систем. Конкретные реализации децентрализованных систем отличаются, в основном, способом маршрутизации сообщений между узлами.

Рассмотрим децентрализованную систему с точки зрения распределения нагрузки.

Хранение информации о состоянии узла

Каждый узел самостоятельно хранит всю информацию о своем состоянии, включая текущий уровень загруженности. Решения о переходах между состояниями принимаются исключительно локально.

Эффективность балансировки нагрузки

В отсутствие центрального компонента каждый узел должен принимать решение об обработке или отказе в обработке входящих запросов только на основании информации, хранящейся на этом узле. Даже если используется алгоритм, обеспечивающий оптимальное использование ресурсов локального узла, с точки зрения всей системы в целом распределение нагрузки может быть далеким от оптимального. С точки зрения клиентов это может приводить к неравномерному распределению ресурсов, когда запросы одних клиентов выполняются быстрее, а запросы других медленнее либо вообще отклоняются независимо от того, в каком порядке они были отправлены на обработку. С точки зрения серверов это может приводить к недогруженности одних серверов в случае, когда другие успевают перехватывать и обрабатывать все приходящие запросы.

Устойчивость

Порядок включения и выключения узлов предполагается произвольным, то есть все узлы изначально проектируются с учетом того, что узел, с которым происходит какое-либо взаимодействие, может отключиться в любой момент. Такой подход требует дополнительных усилий при разработке узла, но в конечном итоге распределенная система обладает очень высокой степенью устойчивости к сбоям или отключениям отдельных узлов. С точки зрения пользователей при отказе в работе отдельных узлов общая производительность системы будет падать, но, тем не менее, система будет оставаться работоспособной, пока в нее присутствует хотя бы один экземпляр каждого компонента, необходимого для решения поставленной прикладной задачи.

Масштабируемость

Процесс добавления нового узла обычно включает в себя рассылку специального уведомления с адресом нового узла. При получении таких уведомлений все заинтересованные узлы обновляют свои локальные таблицы маршрутизации или их аналог, что приводит к тому, что вновь прибывшие узлы сразу начинают получать часть клиентских запросов.

Самоорганизация

Так как реакция на исчезновение узлов из системы и на добавление новых узлов в систему является неотъемлемой частью работы каждого узла, а не какой-то критической ситуацией, большинство децентрализованных систем проявляют свойство самоорганизации. Процесс упорядочивания узлов и конфигурации связей между ними происходит за счет внутренних факторов без какого-либо внешнего управления.

В целом, децентрализованные системы предоставляют большую устойчивость, нежели системы с центральным компонентом, за счет уменьшения общей эффективности использования имеющихся ресурсов. В случае, когда оптимальное распределение и балансировка нагрузки не является критическим требованием, разработка и использовании децентрализованной системы может быть хорошей альтернативе классическим системам с централизованным управлением.

Прототип распределенной системы с децентрализованным распределением нагрузки

Для изучения свойств децентрализованной модели управления нагрузки была разработана гибридная распределенная система. Для примера была выбрана прикладная задача, заключающаяся в рендеринге изображения множества Мандельброта.

Рассмотрим функцию $f_c(z)=z^2+c$ и последовательность (0, $f_c(0)$, $f_c(f_c(0))$, $f_c(f_c(f_c(0)))$, …). Множеством Мандельброта является множество комплексных чисел c, для которых указанная последовательность остается ограниченной.

При генерации изображения множества Мандельброта при помощи компьютера обычно для каждой точки обычно считают некоторое заранее заданное количество итераций. Если последовательность не покинула круг радиуса 2, то точка считается принадлежащей множеству Мандельброта и закрашивается в черный цвет, иначе точка закрашивается либо в белый цвет, либо цвет, зависящим от номера итерации, на которой последовательность покинула круг радиуса 2. Вычисления в каждой точке могут производиться независимо от остального изображения, то есть построение изображения множества Мандельброта является хорошо масштабируемой задачей.

Архитектура прототипа

Разработанная система состоит из компонентов двух типов:

- Клиентское приложение — которое отображает на экране участок изображения множества Мандельброта с выбранным центром и масштабом. В процессе построения изображение разбивается на небольшие куски, называемые тайлами, которые передаются на обработку рендерерам.
- Рендерер — вычисляет цвет каждой точки в тайле, и отсылает готовый тайл обратно на клиентское приложение.

Для упрощения вместо организации полноценной децентрализованной системы был использован гибридный подход, при котором сетевое взаимодействие между узлами, непосредственно необходимое для решения прикладной задачи, осуществляется напрямую при помощи промежуточного программного обеспечения ICE, а служебные сообщения, необходимые для децентрализованного управления нагрузкой, передаются при помощи службы IceStorm.

Единицей работы служит тайл — небольшая область изображения с заданными координатами, размером, масштабом и максимальным количеством итераций. Тайл описывается простой структурой данных, содержащей как исходные параметры тайлы, так и результат вычисления в каждой точке тайла.

```
public class Tile {
  public long x;
  public long y;
  public long scale;
  public int maxIteration;
  public int width;
  public int height;
  public int[] iterationData;
}
```

Клиентское приложение поддерживает очередь тайлов, которые необходимо обработать для того, чтобы построить текущее изображение. Обработанные тайлы возвращаются в очередь и передаются графическому интерфейсу для перерисовки соответствующего участка изображения.

```
public interface TileQueue {
  Tile getTile();
  void submitTile(Tile tile);
}
```

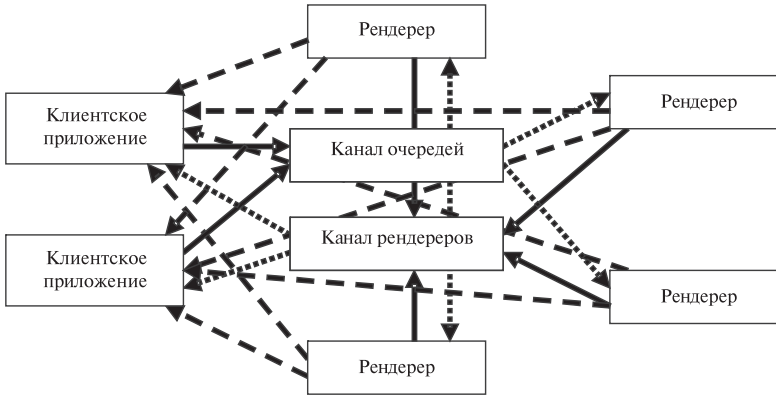

Рис. 2. Прототип децентрализованной системы

После запуска клиентское приложение заполняет очередь тайлами, необходимыми для отрисовки изображения с центром и масштабом, заданными по умолчанию. В дальнейшем, при сдвиге изображения или изменении масштаба, клиентское приложение добавляет в очередь недостающие тайлы, требуемые для отрисовки ранее невидимых участков изображения. После того, как очередь тайлов получит обратно обработанный тайл, клиентское приложение перерисовывает соответствующую область на экране.

Для организации общения между клиентскими приложениями и рендерерами используется два канала сообщений. Первый канал, канал очередей, служит для уведомлений о том, что какая-либо очередь (клиентское приложение) содержит необработанные тайлы. Второй канал, канал рендереров, служит для уведомлений о том, что какой-либо рендерер простаивает, то есть готов к обработке следующего тайла. Каждое клиентское приложение подписано на канал рендереров, а каждый рендерер подписан на канал очередей.

Клиентское приложение и рендерер работают по принципу конечного автомата, переход между состояниями которого может происходить в ответ на получение сообщений из соответствующего канала.

Клиентское приложение содержит четыре процесса, работающих с общей очередью тайлов: процесс, отвечающий за добавление новых тайлов в очередь, процесс, отвечающий за получение обработанных тайлов, процесс, отвечающий за выбор следующего тайла для обработки, и процесс, отвечающий за обработку сообщений из канала рендереров.

Процесс добавления новый тайлов в очередь состоит из следующих шагов:

1. Графический интерфейс запрашивает обработку нового тайла.
2. Тайл помещается в конец очереди необработанных тайлов.
3. В канал очередей отправляется сообщение с адресом данного клиентского приложения.
4. Процесс переходит в состояние ожидания запросов от графического интерфейса.

Процесс получения обработанного тайла состоит из следующих шагов:

1. Клиентское приложение получает обработанный тайл от рендерера (метод submitTile(Tile):void).
2. Тайл удаляется из очереди необработанных тайлов и из очереди обрабатываемых тайлов.
3. Данные тайла передаются в графический интерфейс.
4. Процесс переходит в состояние ожидания обработанных тайлов.

Процесс выбора следующего тайла для обработки состоит из следующих шагов:

1. Клиентское приложение получает запрос от рендерера на обработку тайла (метод getTile():Tile).
2. Если очередь необработанных тайлов не пуста, то из ее начала извлекается тайл, и процесс переходит на пункт 5.
3. Если очередь обрабатываемых тайлов не пуста, то из ее начала извлекается тайл, и процесс переходит на пункт 5.
4. Рендереру отправляется ответ, что очередь пуста, и процесс переходит в состояние ожидания.
5. Выбранный тайл помещается в конец очереди обрабатываемых тайлов.
6. Выбранный тайл отправляется рендереру.
7. Процесс переходит в состояние ожидания запросов от рендереров.

Использование дополнительной очереди тайлов, уже отправленных на обработку, обеспечивает гарантированную обработку всех тайлов даже в том случае, когда какой-либо рендерер после получения тайла неожиданное прекращает работу. В этом случае тайлы, остающиеся в очередь уже отправленных на обработку, будут повторно рассылаться другим рендерерам.

Процесс обработки сообщений из канала рендереров состоит из следующих шагов:

1. Клиентское приложение получает из канала рендереров сообще-ние о том, что есть свободный рендерер.
2. Если или очередь необработанных тайлов не пуста, или очередь уже отправленных на обработку тайлов не пуста, то в канал очередей отправляется сообщение с адресом данного клиентского приложения.
3. Процесс переходит в состояние ожидания сообщений из канала рендереров.

Клиентское приложение реализовано в двух вариантах. Первый вариант представляет собой графическое приложение на Java, а второй представляет собой веб-приложение. В обоих случаях поддерживается изменение масштаба и сдвиг изображения.

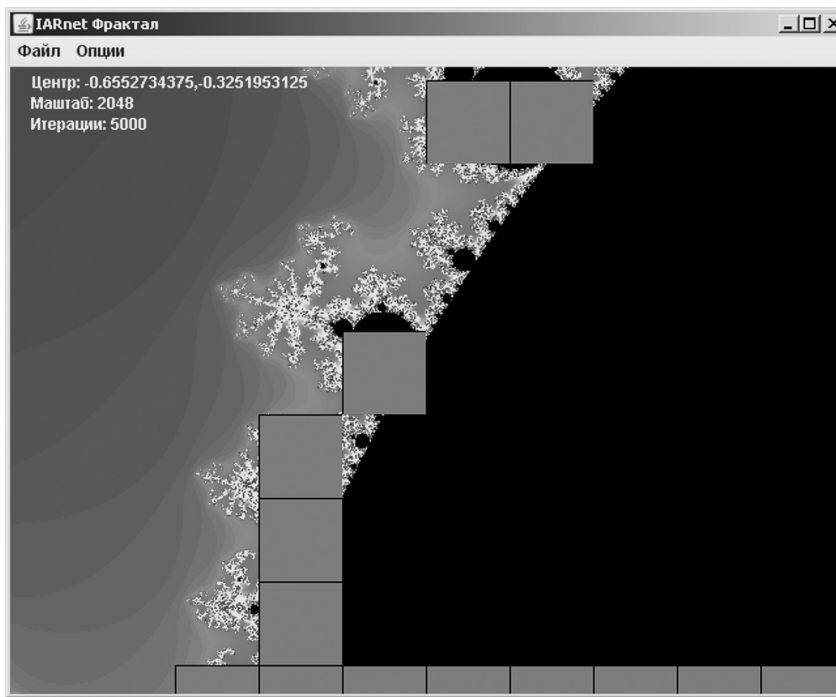

Рис. 3. Клиентское приложение в процессе построения изображения

Рендерер содержит один процесс, который выполняет все действия, необходимые для обработки тайла.

1. Рендерер получает сообщение из канала очередей о том, что есть непустая очередь.
2. Рендерер вызывает метод getTile():Tile на очереди, указанной в сообщении.
3. Если рендерер получил ответ, что очередь пуста, то процесс переходит в состояние ожидания.
4. Рендерер обрабатывает полученный тайл.
5. Рендерер отсылает обработанный тайл обратно при помощи метода submitTile(Tile):void.
6. В канал рендереров отправляется сообщение о том, что есть свободный рендерер.
7. Процесс переходит в состояние ожидания сообщений из канала очередей.

Кроме того, каждый рендерер после запуска так же отправляет в канал рендереров сообщение о том, что есть свободный рендерер. Это необходимо для того, чтобы имеющиеся в системе клиентские приложения могли среагировать на появление нового рендерера и загрузить его работой.

Рендерер реализован в виде консольного приложения на Java.

Выводы

Основные преимущества децентрализованного распределения нагрузки по сравнению с централизованным балансировщиком вытекают из общих свойств децентрализованных систем: высокая устойчивость и масштабируемость, самовосстановление после сбоев. В то же время децентрализованная система обладает рядом недостатков, которые необходимо учитывать при выборе архитектуры распределенного приложения в целом или при выборе подхода к балансировке нагрузки:

- При децентрализованном управлении нагрузкой очень сложно достичь оптимального распределения нагрузки по имеющимся узлам. Это означает, что для достижения того же уровня производительности децентрализованная система должна иметь большее количество узлов, чем система с централизованным управлением.

- Архитектура и модель поведения узлов, используемых в децентрализованной системе, существенно отличается от архитектуры и модели поведения обычных серверных приложений. Шаблоны проектирования, используемые в обычных централизованных системах, практически не применимы при разработке узлов децентрализованной системы.

Кроме того, разработанный прототип зависит от сервиса рассылки сообщений IceStorm, который используется для обнаружения новых узлов и маршрутизации сообщений между узлами. Сбой в работе сервиса рассылки сообщений влечет за собой сбой в работе всей системы. Для настоящей децентрализованной системы необходимо создание собственного механизма маршрутизации сообщений, встроенного в каждый узел. Подобная функциональность не зависит от конкретной прикладной задачи, решаемой в распределенной системе, и может быть реализована в виде библиотеки, пригодной для повторного использования.

Литература

1. *Michi Henning, Mark Spruiell*. Distributed Programming with Ice. Revision 3.2.1, August 2007 / http://zeroc.com/doc/Ice-3.2.1/manual
2. Mandelbrot set / http://en.wikipedia.org/wiki/Mandelbrot_set