Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Санкт-Петербургский Государственный Университет» (СПБГУ)

Математико-механический факультет

Кафедра системного программирования

Обеспечение надёжности и высокой доступности кластера СХД.

Курсовая работа студента 445 группы

Котова Юрия Александровича

Научный руководитель ……. С. В. Богатырев

Санкт-Петербург

2011

Содержание

Введение

Во многих организациях главным активом является информация, поэтому нужно обеспечить надежное хранение информационных ресурсов и быстрый доступ к ним. На данный момент существует множество решений, но на небольших предприятиях они являются редкостью из-за слишком высокой стоимости. Целью проекта является создание надежной системы хранения данных, работающей на дешевом, доступном оборудовании и обладающей следующими свойствами:

- высокая доступность;
- масштабируемость;
- отказоустойчивость;
- простота администрирования.

Для обеспечения этих требований используются кластерные системы.

Кластер – это две или более самостоятельные системы, соединенные в единую систему высокого уровня доступности посредством специального программного и аппаратного обеспечения и представляющие с точки зрения пользователя единый аппаратный ресурс.

Задача кластера заключается в обеспечении согласованной работы всех узлов для достижения поставленной цели. Целью может быть высокая устойчивость (HA – High Availability), высокая вычислительная способность (HP – High Performance), параллельное вычисление, параллельное обслуживание запросов. Кластер отличается от клиент-серверных распределенных вычислений тем, что кластерное взаимодействие – это равноценное взаимодействие каждого узла друг с другом по принципу пиринговых сетей (peer-to-peer), когда каждый узел является членом более крупной системы. Условно, в кластере выбирается мастер-узел, а другие являются «подшефными» узлами. В случае отказа мастер-узла — по заранее выбранному алгоритму выбирается новый мастер-узел, который и курирует в дальнейшем весь кластер.

Постановка задачи

Данная работа выполняется в рамках проекта «Cirrostratus», который представляет собой сеть хранения данных с фиксированной топологией, поэтому остро стоят проблемы отказоустойчивости и перераспределения нагрузки. Основной целью данной работы является модернизация проекта «Cirrostratus» из сети в кластер хранения данных, отвечающий следующим характеристикам:

- отказоустойчивость — возможность продолжать работу в полном объёме в случае выхода из стоя любого узла, нет единой точки отказа;
- доступность — динамическая балансировка нагрузки между узлами системы;
- масштабируемость — гибкое подключение узлов к кластеру.

Для достижения поставленных целей необходимо выполнить следующие задачи:

1. ***Обзор существующих решений и реализаций систем управления ресурсами кластера;***
2. ***Выбор наиболее подходящего решения;***
3. ***Разработка ресурсных агентов для взаимодействия системы с управляющим ПО кластера;***
4. ***Оптимизация параметров ресурсных агентов и настроек менеджера ресурсов для достижения оптимальной производительности и надёжности.***

Обзор кластерных решений

Сначала рассмотрим способы балансировки нагрузки между узлами (серверами) системы. Первый вариант: с использованием **Round robin DNS (RR DNS)** — метода, позволяющего «размазывать» запросы между n-ым количеством серверов, отдавая на каждый DNS запрос новый IP из ранее определённого пула адресов. Хотя Round robin DNS легко реализовать, всё же этот алгоритм имеет несколько проблематичных недостатков:

- *Сложно управлять* — задается группа IP-адресов, нет управления весами, не отслеживается состояние серверов (при выходе сервера из строя RR DNS продолжит раздавать его IP-адрес). Фактически, размазываются запросы по диапазону IP-адресов, но не балансируется нагрузка на серверах;
- *Кэширование* клиента, по большому счёту, становится просто невозможным.

Существуют методы, позволяющие преодолеть некоторые ограничения. Например, модифицированные DNS-сервера (такие, как lbnamed) могут регулярно опрашивать зеркала серверов для проверки их доступности и нагруженности. Если сервер не отвечает, то по мере необходимости он может быть временно удален из пула DNS, пока не сообщит, что опять работает в соответствии со спецификацией. Но проблема кэширования остаётся актуальной, поэтому перейдём к следующему способу — LVS.

Linux Virtual Server (**LVS**) — широко распространённое средство управления кластерных системам для Linux систем, с помощью которого можно построить надёжный и высокоскоростной сервер, основываясь на кластерной технологии. Фактически, это модуль ядра Linux (ipvs).

LVS представляет из себя L4-роутер, позволяющий прозрачно и управляемо раздавать пакеты по заданным маршрутам.

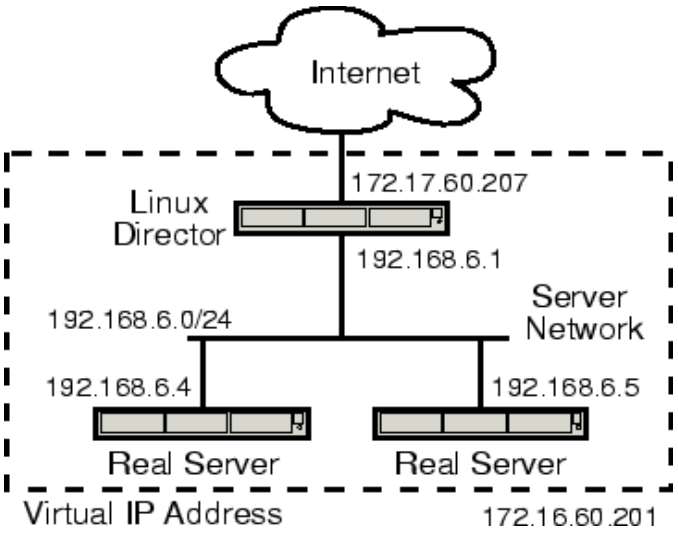

Рис.1 LVS топология

Основная терминология:

- **Director** — узел «директор», «балансировщик», осуществляющий роутинг.
- **Real server** — узел «сервер», «обработчик» нашей системы, обрабатывающий пакеты.
- **VIP** или Virtual IP — IP собранного кластера, показанный во внешний мир.
- **DIP** и **RIP** — IP директора и серверов соответственно.

На директоре включается модуль ядра IPVS (IP Virtual Server), настраиваются правила раздачи пакетов серверам и поднимается VIP — обычно как дублер (алиас) к внешнему интерфейсу. Пользователи видят систему через VIP как единое целое. Пакеты, пришедшие на VIP, передаются выбранным методом одному из Real server'ов и уже там отрабатываются. Клиенту кажется, что он работает с одной машиной.

Правила передачи пакетов крайне просты: задается виртуальный сервис, определяемый парой VIP:port. Сервис может быть TCP или UDP. Здесь же задается метод ротации узлов (планировщик). Далее задаем набор серверов из нашей системы также парой RIP:port, а также указываем метод передачи пакетов и вес узла, если того требует выбранный планировщик. Чем выше вес сервера, тем с большей вероятностью он получит запрос. Если выставить вес в 0, то сервер будет исключен из всех операций — это очень удобно, если нужно вывести сервер из эксплуатации.

По умолчанию пакеты передаются методом DR. Существуют следующие варианты роутинга:

- **Direct Routing** (gatewaying) — наиболее простой метод, пакет направляется напрямую к обработчику, без изменений. При этом возникают проблемы восприятия пакета обработчиком, т.к. в пакете указан VIP и обработчик распознаёт его как не предназначающийся машине. При смене порта необходима настройка IP таблиц.
- **IPIP incapsulation** (tunneling) — инкапсуляция **IPIP (**туннелирование) — модификация предыдущего метода на тот случай, когда балансировщик и обработчики находятся в разных сегментах сети. При помощи инкапсуляции IP-в-IP создаются сетевые туннели, которые объединяют балансировщик и обработчик в один сегмент виртуальной сети.
- **NAT** (masquarading) — наиболее гибкое и универсальное решение: балансировщик подменяет адреса/порты назначения проходящих пакетов на реальные адреса/порты обработчиков (трансляция порт-адресов). При этом обеспечивается как простота настройки (не нужно поднимать дополнительные адреса на обработчиках и туннели между балансировщиком и обработчиком), так и гибкость (целевой порт обработчика может отличаться от порта кластерной службы, что невозможно в рамках других методов). К недостаткам данного метода можно отнести то, что маршрут по умолчанию у всех обработчиков направлен на балансировщик

ровщика, что не всегда удобно.

LVS предоставляет довольно общирный выбор планировщиков:

- **Persistent connection** - постоянное соединение;
- **rr (round robin)** — классическая кольцевая схема, когда все обработчики получают новые соединения поочередно;
- **wrr (weighted round robin)** — модификация кольцевой схемы на тот случай, когда одни сервера могут обрабатывать больше соединений, чем другие. Такие сервера включаются в один проход очереди по нескольку раз и, соответственно, получают больше соединений;
- **lc (least-connection)** — новое соединение передается тому серверу, который на текущий момент обрабатывает наименьшее число соединений;
- **wlc (weighted least-connection)** — модификация предыдущего метода на случай, когда возможности серверов отличаются. При выборе нового сервера сравниваются не непосредственно количества открытых соединений, а отношения количества соединений к весовому коэффициенту (весу) обработчика (который и характеризует его возможности);
- **lblc (locality-based least-connection)** — формирует привязку обработчиков к кластерному адресу назначения (их может быть несколько). В том случае, если для запрошенного адреса такой привязки еще нет или привязанный обработчик сильно перегружен — выбирает (по алгоритму lc) и привязывает новый обработчик;
- **dh (destination hashing)** — обеспечивает жесткую, неуправляемую привязку обработчиков к кластерным адресам назначения (выбор обработчика производится исключительно на основании элементарных арифметических операций над кластерным адресом и никак не учитывает загруженности обработчиков). Может привести к перегрузкам отдельных обработчиков — в этом случае кластер просто перестает обслуживать соответствующий адрес. В такой ситуации нужно либо увеличить лимит соединений на обработчик, либо выбрать другой алгоритм.
- **sh (source hashing)** — аналогично dh, но обработчики привязываются к адресам клиентов. Как и dh, в случае перегрузки прекращает обслуживание. Отметим, что для формирования временных привязок обработчиков к адресам клиентов целесообразнее использовать механизм устойчивых соединений (persistent connections), доступный для всех алгоритмов.

В рассмотренной выше конфигурации LVS очевидной точкой отказа, вызывающей разрушение всего кластера, будет директор. На этот случай, ipvsadm поддерживает запуск в режиме демона с возможностью синхронизации таблиц и текущих соединений между

несколькими директорами. Один, очевидно, станет мастером (active), остальные будут слейвами (backup) директорами. Но потребуется так же переместить VIP на backup директор в случае отказа active. Тут на помощь приходят HA решения, которые будут рассмотрены далее. Другой задачей будет мониторинг и своевременный вывод из эксплуатации серверов из нашей системы — проще всего это делать весами.

Кластер HA или кластер высокой готовности представляет собой набор серверов, которые совместно работают для предоставления определенных сервисов. Сервисы принадлежат не конкретному серверу, а всему кластеру. Если происходит сбой одного из серверов, его функции автоматически переходят к другому серверу кластера и все сервисы, предоставляемые кластером, продолжают работу. Аналогично, в случае отказа сервиса на одном из кластерных серверов, он будет автоматически запущен на другом сервере. Чтобы диагностировать сбои в работе кластера необходимо следить за состоянием серверов, для этого применяется механизм сердцебиения. Этот механизм немного напоминает своей работой процесс Linux init, но для всего кластера. Он отвечает за запуск и останов сервисов таким образом, что каждый сервис в один момент времени выполняется где-то в кластере.

Механизм сердцебиения использует скрипты аналогично стандартному init для запуска и останова сервисов. Управление ресурсами происходит на основе групп. Ресурсы одной группы всегда выполнятся на одной и той же системе в кластере. Группы ресурсов задаются в конфигурационных файлах.

Есть несколько реализаций с открытым кодом механизма сердцебиения для Linux кластеров: keepalived, heartbeat, corosync.

Keepalived – это демон, осуществляющий мониторинг доступности сервера в кластере. При отказе сервера демон исключит его из LVS пула. Так же keepalived позволяет поддерживать передачу прав владения общим IP-адресом между узлами кластера. Но он не может мониторить и управлять сторонними сервисами, не относящимися к LVS и для него нет расширения в виде системы управления ресурсами.

Heartbeat – это один из общедоступных пакетов проекта с открытыми исходными кодами High-Availability Linux (Linux-HA). Он обеспечивает основные функции, требующиеся для любой HA-системы, например, запуск и останов сервисов, мониторинг доступности сервера в кластере и передача прав владения общим IP-адресом между узлами кластера, следит за состоянием конкретного сервиса (или сервисов) по последовательному кабелю, интерфейсу Ethernet либо по обоим. Heartbeat предоставляет фундамент для более сложных сценариев, например, конфигурации активный/активный, в которой оба узла работают параллельно и распределяют нагрузку. Heartbeat может быть дополнен одним из менеджером ресурсов: mon, lddirector, Pacemaker. Первые два представляют собой интерфейс к функциям heartbeat и, в первую очередь, используются для поддержки работоспособности LVS, их конфигов.

Pacemaker же это полноценный менеджер ресурсов, он позволяет управлять любыми (настроенными) сервисами, вводить зависимости между. Более детальное рассмотрение последует далее. Однако Pacemaker рекомендуется использовать на основе пакета corosync.

Corosync — это общедоступный пакет с открытыми исходными кодами проекта OpenAIS. Он предоставляет аналогичную функциональность пакета Heartbeat, но он лучше совместим с проектом Pacemaker и на его основе легче строить кластера, состоящие из множества узлов.

Исходя из перечисленных факторов, для построения кластера было выбрано сочетание: *Corosync + Pacemaker*. Далее рассмотрим более детально, как работает менеджер ресурсов Pacemaker.

Pacemaker

Pacemaker — набор утилит от ClusterLabs для управления распределением ресурсов кластера. Его основные свойства:

- Обнаружение и восстановление сбоев на уровне узлов и сервисов;
- Независимость от подсистемы хранения: общий диск не требуется;
- Независимость от типов ресурсов: все что может быть заскриптовано, может быть кластеризовано;
- Поддержка STONITH (Shoot-The-Other-Node-In-The-Head)
- Поддержка кластеров любого размера;
- Поддержка практически любой избыточной конфигурации;
- Автоматическая репликация конфига на все узлы кластера;
- Возможность задания порядка запуска ресурсов, а также их совместимости на одном узле;
- Поддержка расширенных типов ресурсов: клонов (запущен на множестве узлов) и с дополнительными состояниями (master/slave и т.п.);
- Единый кластерный шелл (crm), унифицированный, скриптующийся.

Pacemaker представляет из себя конгломерат из пакетов Pacemaker, Corosyc, OpenAIS, Heartbeat. Для взаимодействия между узлами кластера Pacemaker может использовать либо Corosync/OpenAIS, либо Heartbeat. Несмотря на то что предпочтительным является Corosync, пакет Heartbeat так же должен быть установлен, поскольку он включает в себя достаточно большое количество OCF скриптов для управления ресурсами кластера.

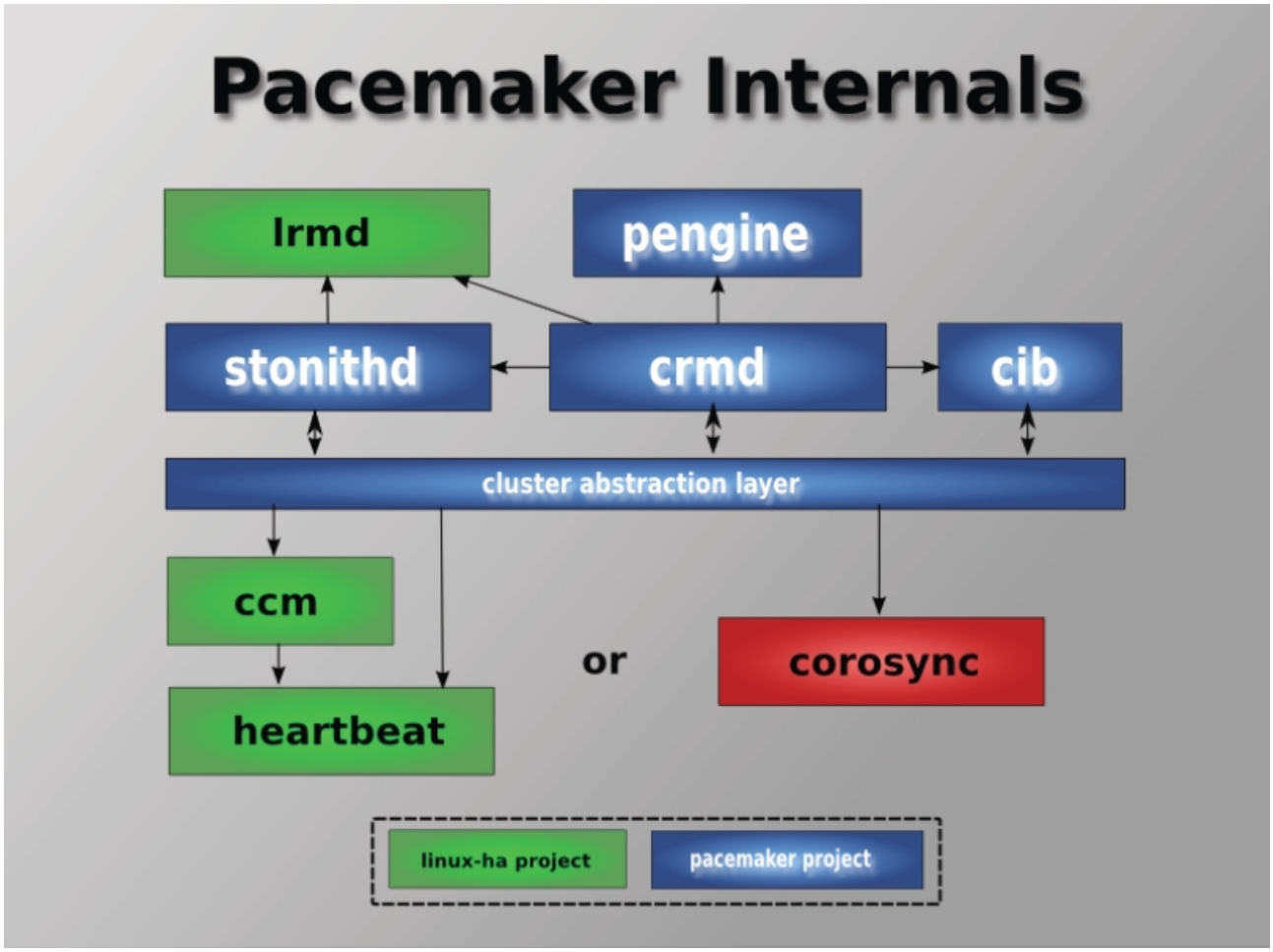

Рис.2 Устройство Pacemaker

Основные понятия:

Узлы (ноды,nodes) кластера

Узел (нода,node) кластера представляет из себя машину с установленным Pacemaker и включенным в состав кластера. Узлы, предназначенные для выполнения одинаковых ресурсов, должны иметь одинаковую конфигурацию софта, который используется ресурсами. Это исключает ситуацию, когда ресурс при перемещении между узлами может запуститься на одном узле и не может запуститься на другом.

Ресурсы

С точки зрения pacemaker, ресурс — это скрипт, умеющий выполнять 3 действия: start, stop и monitor и соответствующие LSB (Linux Standard Base) или OCF (Open Cluster Framework) — последнее несколько расширяет LSB, требуя также передачи параметров через переменные окружения с особым названием. Обычно скрипты пишутся на bash, но могут быть написаны на Perl, Python или даже на C.

Ресурс может представлять из себя:

- IP адрес
- Демон с определенной конфигурацией
- Блочное устройство
- Файловую систему
- etc

Группы ресурсов

Это последовательный список ресурсов, которые должны запускаться в определенном порядке, останавливаться в обратном порядке и исполняться на одном узле.

Связи

Связи определяют привязку ресурсов к узлу (**location**), порядок запуска ресурсов (**ordering**) и совместное их проживание на узле (**colocation**). Для установки предпочтений связей используется **вес связей** от -INFINITY до +INFINITY. Установка с весом -INFINITY или +INFINITY является жесткой. Вес +/- INFINITY не может оспариваться другими условиями, а в целочисленном представлении может.

Таким образом можно строить довольно сложные цепочки зависимостей ресурсов. Все становится еще интереснее, если учесть, что можно задавать более сложные правила (rules), срабатывающие, например, только в определенное время суток или в зависимости от состояния компонентов кластера.

Ресурсный агент

Для взаимодействия нашего программного обеспечения (демона), обслуживающего систему хранения данных, и менеджера, управляющего кластером, надо, что бы Pacemaker распознавал нашего демона как свой ресурс. Для этого необходимо написать ресурсный агент. Ресурсный агент — это стандартизированный интерфейс для ресурса кластера. Он переводит стандартный набор операций в конкретные шаги для приложения и интерпретирует их результаты как успех или неудачу.

Наш ресурсный агент — это скрипт формата OCF (Open Cluster Framework). Он обрабатывает следующие действия:

- **Start** – стартует ресурс с параметрами переданными через переменную окружения OCF_RESKEY_<param> . При удачном старте start возвращает OCF_SUCCESS, а в противном случае OCF_ERR_GENERIC;
- **Stop** – останавливает ресурс;
- **Monitor (status)** – проверяет работу ресурса;
- и другие служебные: usage, meta_data, reload.

Влияние таймаутов распространяется не на ресурсный агент, а на менеджера ресурсов кластера, который проверяет насколько долго ресурс запускается либо выполняет какое-либо другое действие, и завершает его, если ответ не был получен за определённое время. Поэтому ресурсы не должны сами проверять любые таймауты. Однако, ресурсные агенты могут давать советы по использованию разумных значений таймаутов.

Каждое действие, обрабатываемое ресурсным агентом, должно иметь своё значение таймаута, используемое по умолчанию. Таймаут используется для того, чтобы определить, что ресурс не работоспособен, если агент не ответил на какое-либо действие за время таймаута для этого действия.

Кроме того повторяющиеся действия, такие как monitor, должны иметь рекомендованный интервал между двумя последовательными вызовами действия.

Далее мы рассмотрим параметры, влияющие на работу кластера и, в частности, выбор таймаутов и интервалов.

Параметры кластера

«Идеальная» система хранения данных работает так, что пользователь всегда имеет доступ к данным вне зависимости от того, что происходит внутри системы. Мы хотим, чтобы наша система была «идеальной», чтобы пользователь не замечал выхода из строя ресурса и даже целого узла системы.

Для обеспечения этого надо, чтобы время ответа пользователю не превышало времени ожидания ответа от системы. В нашей системе мы используем протокол AoE и пользователь может установить произвольное время ожидания ответа, но мы знаем, что в модуле ядра Linux aoe таймаут по умолчанию равен 180 секунд. Будем ориентироваться на данное время.

Мы можем варьировать следующими параметрами:

1. Частота, с которой узлы кластера опрашивают друг друга о том, что они живы.
2. Таймауты по умолчанию для каждого действия ресурса.
3. Интервал мониторинга ресурса.

Посмотрим, как эти параметры влияют на работоспособность кластера. Для начала рассмотрим это в предположении, что при выходе из строя узла или ресурса ресурс гарантированно запустится на другом узле. В этом случае максимальное время простоя ресурса будет равно сумме максимального времени диагностирования неработоспособности ресурса и времени на запуск ресурса на другом узле. Ресурс нашей системы запускается достаточно быстро и, в сравнении с временем диагностики, оно очень мало, поэтому его можно не рассматривать.

Посмотрим, как получается максимальное время диагностики неработоспособности ресурса. Максимальное время диагностики при выходе из строя узла кластера характеризуется частотой с которой узлы кластера опрашивают друг друга о том, что они живы. При выходе из строя ресурса системы максимальное время диагностики будет равно интервалу мониторинга ресурса. Итак, получаем:

$$T_{простоя} = \max\{T_{узла};\ T_{монитор} + T_{таймаут\_монитор}\}$$

Где:

- $T_{простоя}$ - максимальное простоя ресурса
- $T_{узла}$ — интервал мониторинга узлов
- $T_{монитор}$ — интервал мониторинга ресурса
- $T_{таймаут\_монитор}$ — таймаут для действия monitor

Однако предположение о гарантированном запуске ресурса на другом узле достаточно сильно ограничивает надёжность кластера. Для обеспечения более высокой надёжности необходимо рассматривать случаи ошибки при запуске ресурса на другом узле и выхода из строя узла после запуска ресурса. В таком случае формула будет выглядеть так:

$$T_{простоя} = \max\{T_{узла};\ T_{монитор} + T_{таймаут\_монитор}\} + \max\{T_{узла};\ T_{таймаут\_старт}\}$$

Подбирать параметры надо таким образом, чтобы $T_{простоя}$ было меньше времени ожидания ответа пользователем от системы. Когда данные о пользователе не известны, можно ориентироваться на 3 минуты.

В нашего ресурсного агента используются следующие значения:

- $T_{монитор}$ = 10 секунд (интервал мониторинга ресурса)
- $T_{таймаут}$ = 15 секунд (таймаут для действия любого)

Заключение

В данной работе был проведён обзор существующих решений для построения надежного и высоко доступного кластера. Так же был проведён обзор реализаций систем управления ресурсами кластера, было выбрано наиболее подходящее решения для проекта «Cirrostratus», разработан ресурсный агент для взаимодействия менеджера ресурсов и нашей системы, были выявлены параметры и проанализировано их влияние на работу кластера, выбраны конкретные параметры для проекта «Cirrostratus».

В результате данной курсовой работы была ликвидирована единая точка отказа в системе хранения данных проекта «Cirrostratus».

Список литературы

[1] Alan Robertson. The Open Cluster Framework (OCF).

[2] Dejan Muhamedagic, Yan Gao. CRM CLI (command line interface) tool.

[3] Kai Uhlemann. High Availability for High-End Scientific Computing. //NETWORK CENTERED COMPUTING, HIGH PERFORMANCE COMPUTING AND COMMUNICATION, FACULTY OF SCIENCE, THE UNIVERSITY OF READING (14. March 2006)

[4] Lars Marowsky-Brée. The Open Clustering Framework. //Ottawa Linux Symposium (June 26th–29th, 2002)