1 2003 год

1.1 Общее описание OpenGFS

1.1.1 Особенности

OpenGFS — журналируемая кластерная файловая система, поддерживающая одновременное использование общего дискового накопителя несколькими узлами. Некоторые особенности, которые стоит отметить:

- Каждый узел имеет прямой доступ к устройству. На всех системах дисковый ресурс должен быть представлен одним именем в /dev/ и иметь одинаковый размер. Это обеспечивает более эффективное использование канала, так как не требуется передавать служебные данные. Также это позволяет отказаться от центрального сервера, который обычно становится самым узким местом кластерной файловой системы.

- Наличие системы блокировок, контролирующих одновременный доступ к файлам. Поддерживается несколько протоколов блокировки (таких как OpenGFS 'memexp' и 'nolock'). И, хотя есть некоторые трудности с обработкой большого числа легковесных операций записи, это обеспечивает разрешение конфликтов при одновременном доступе.

- Индивидуальные журналы. В процессе работы каждому узлу приписывается журнал, что позволяет откатить операцию записи, если в процессе ее выполнения произошел сбой, даже критический. Использование разными компьютерами независимых журналов гарантирует сохранение их целостности. Журналы могут быть как внутренними — внутри файловой системы, так и внешними — расположенными на независимом дисковом накопителе.

- Система координации, обеспечивающая выдачу журнала новым узлам, когда те присоединятся к кластеру, и восстановление целостности данных, когда узел аварийно завершает свою работу.

- Поддержка оключения от кластера вышедших из под контроля компьютеров. OpenGFS поддерживает некоторе количество устройств управления питанием (STOMITH) и разграничения доступа (Fencing), которые в аварийном случае должны либо выключить узел, либо отрезать его доступ к общему диску. В связи с тем, что не было доступа к подобного рода оборудованию, тестирование этой возможности проведено не было.

- Изменяемый размер. В процессе работы можно добавлять новые журналы и увеличивать вместимость файловой системы, если в системе появятся новые ресурсы. К сожалению, на данный момент не поддерживается уменьшение выделенного пространства и количества журналов. Изменение размера файловой системы возможно без остановки ее работы, однако это должно поддерживать ни-

жележащее блочное устройство, такое как Enterprise Volume Management System (EVMS), Logical Volume Manager (LVM) и Device Mapper (DM) .

1.1.2 Приложения

Возможности и ограничения GFS являются следствием потребностей таких приложений, как сервера баз данных, систем анализа геофизических измерений и обработки компьютерной графики. Так как OpenGFS является преемником ранних версия GFS (до 4.x), она сохранила стуктуру и, соответственно, многие достоинства и недостатки предшественника. Приложения, для которых разработна GFS, объединены одним общим свойством — им необходимо иметь доступ к большим массивам данных с разных вычислительных узлов. Кроме этого, им требуется, чтобы этот доступ был надежным и защищал данные от повреждения — даже при выходе из строя некоторых точек доступа. Этим требованиям GFS полностью удовлетворяет — поддержка систем отсечения вышедших из под контроля узлов вместе с журналированием позволяет избежать потери целостности, а разделение журналов позволяет не создавать систем их мониторинга.

1.2 Структура

OpenGFS логически разделена на несколько модулей — модули блокировок и непосредственно файловой системы. Последний так же ответсвеннен за кеширование данных. В ранних версиях OpenGFS присутвовали модули управления виртуальными разделами, но в последних от этого отказались — появились специализированные системы управления (например LVM), которые позволяют добиться лучших результатов. Сама файловая система разделена на журналы, как внешние так и внутренние, раздел кластерной информации и собственно данные. Журналы отвечают за целостность данных на дисковом накопителе и позволяют откатить, либо успешно завершить, некоторые изменения, которые не были доведены до конца. Раздел информации содержит данные о подключенных узлах, именах устройств и серверу синхронизации доступа.

1.2.1 Блокировки

Модуль блокировок ответственнен за корректное разграничение доступа на запись к блокам данных. Подразумевается, что одновременная запись в разные блоки данных не вызывает конфликтов синхронизации на общем диске. Исходя из этого, модуль блокировок закрывает доступ к блоку данных до тех пор, пока не закончится предыдущая операция. Однако это не всегда так — например, использование распределенного RAID5 не гарантирует отсутствия конфликтов при одновременной записи в разные блоки. Это и стало основной причиной неудачи с такой системой.

На данный момент OpenGFS поддерживает два протокола блокировок - nolock и memexp. Первый предусмотрен для работы в однопользовательском режиме. Он превращает OpenGFS в обычную файловую систему. Второй, memexp, — это сетевой

протокол блокировок. Выделяется один узел, на котором запускается memexpd. Он отвечает за синхронизацию запросов на доступ и может быть настроен как на работу с памятью, так и с диском. Хранение таблицы блокировок на жестком диске несколько замедляет его работу, однако повышает надежность файловой системы и в идеале позволяет пережить перезапуск демона синхронизации.

1.2.2 Кеширование

Так как OpenGFS отделена от транспортного уровня и спроектирована под общее внешнее устройство, кеширование данных практически отсутствует — только на уровне dentry. Это обусловлено тем, что данные на носителе быстро меняются, и нет возможности предугадать, когда эти изменения произойдут. Однако в OpenGFS, как и в любой другой файловой системе, существует стандартный механизм кеширования — dentry. Он присутсвует во всех файловых системах и необходим для ускорения обработки деревьев каталогов. Однако данный механизм не кеширует данные файлов. Эта возможность не реализована в OpenGFS, так как за нее отвечает непосредственно ядро операционной системы Linux. Общее для всех файловых систем кеширование не зависит от транспортного уровня и обеспечивает унифицированный механизм работы с копиями данных в памяти. Кеширование операционной системы позволяет использовать не изменившиеся данные из локальной копии в памяти, и, хотя было спроектировано для работы с жесткими дисками, вполне подходит для работы с сетевыми файловыми системами.

1.3 Общее блочное устройство

1.3.1 Внешние дисковые накопители

GFS разрабатывалась для использования с распределенным общим устройством, таким, как общий внешний дисковый накопитель. Протокол iSCSI позволяет использовать один внешний диск на большом количестве вычислительных узлов. Объединение дисковых ресурсов и их потребителей в высокоскоростную сеть — SAN (Storage Area Network) дает возможность эффективно использовать имеющиеся диски, в частности, менять вышедшие из строя диски, практически не нарушая работу системы, подключать несколько ресурсов. Использование специализированного оборудования (свичей и серверов) позволяет добится высокой производительности системы вкупе с освобождением вычислительных систем от работы с диском. Архитектура с общим внешним хранилищем данных довольно легко масштабируется, правда в некоторых пределах — пока будет хватать пропусконой способности транспортного уровня.

1.3.2 Построение общего устройства

Отделение файловой системы от транспортного уровня повышает гибкость, но в то же время ставит задачу построения общего диска (далее Shared Block Device). На тестовом кластере отведенное под построение OpenGFS дисковое пространство было рав-

номерно распределено по узлам — по 60Gb на каждом. Это является довольно распространенной моделью расположения дисковых ресурсов. Таким образом, каждый узел должен экспортировать свой диск и импортировать диски оставшихся узлов. Данная возможность поддерживается в ядре Linux — Network Block Device (nbd). Поверх этих ресурсов может быть построен програмный рейд (Software RAID). Таким образом, на узле будет создано устройство, использующее все предоставленные кластером ресурсы. Если взят RAID с избыточной информацией, такой как RAID5, то этот ресурс может выдержать выход из строя одного узла. Кроме этого может быть подключено некоторое количество дополнительных дисков, чтобы в случае сбоя восстановить избыточность и быть готовым к выходу из строя очередного узла. Целостность информации обеспечивается механизмами самой файловой системы, так что отпадает необходимость разрешать конфликты уровня дискового накопителя. Однако в случае RAID с избыточной информацией возможны проблемы с синхронизацией записи так называемой «parity information», отвечающей за возможность восстановления после выхода из строя одного диска.

1.4 Реализация

В качестве сетевого блочного устройства, транспортного уровня всей системы, был выбран enbd (Enchanced Network Block Device). Несмотря на то, что enbd имеет не самую высокую производительность (17M/c по SCI соединению), его использование обусловлено, в частности, простотой и возможностью не пересобирать ядро системы. Кроме этого, так как большая часть работы происходит в пользовательском процессе, а не в ядре, enbd позволяет довольно легко преодолевать сбои отдельных частей системы.

Из рассмотренных в ходе работы альтернативных вариантов хотелось бы выделить iSCSI сервер (UNH-iSCSI). Эта разработка, позволяет эмулировать внешний iSCSI диск с помощью компьютера под управлением ОС Linux. Протокол iSCSI позволяет использовать общие диски с доступом по оптоволокну или иному высокоскоростному транспорту, однако на данный момент наибольшую популярность завоевали именно оптичволоконные системы. Однако заставить его корректно работать без SCSI дисков не удалось. Встроенная в него возможность использовать в качестве диска обычный файл, видимо, сделана для тестовых целей. На файлах большого размера (более 2Gb) на жестких тестах происходили системные сбои.

1.5 Результаты тестов

1.5.1 RAID-0

В конфигурации с двумя узлами и RAID0 скорость последовательного чтения превосходила 30M/c. Два параллельно запущеных теста на разных узлах - по 15M/c. Ниже приводятся некоторые результаты для 120Gb общего диска с SCI соединением между узлами.

	Один тест	CPU	Два теста параллельно	CPU
Запись посимвольно	6000 K/s	66%	500 K/s	11%
Запись блочная	5500 K/s	34%	2800 K/s	8%
Перезапись	6000 K/s	21%	1500 K/s	11%
Чтение посимвольно	16000 K/s	79%	1200 K/s	60%
Чтение блочное	32000 K/s	18%	15000 K/s	8%

Так же были сделаны замеры для системы из двух узлов, соединенный между собой сетью Ethernet 100Mbit/s.

	Один тест	CPU	Два теста параллельно	CPU
Запись посимвольно	6700 K/s	51%	300 K/s	8%
Запись блочная	6600 K/s	17%	2100 K/s	7%
Перезапись	3550 K/s	10%	2200 K/s	6%
Чтение посимвольно	10000 K/s	49%	5300 K/s	26%
Чтение блочное	11200 K/s	5%	6700 K/s	4%

Хорошо видно, что производительность OpenGFS/ENBD резко падает при большом количестве легковесных операций записи. Это объясняется тем, что происходит очень много блокировок, и пользователь вынужден ждать, пока ему будет позволено начать запись. Кроме этого, большое количестов запросов перегружают канал передачи и, таким образом, увеличивается время на обработку каждого.

1.5.2 RAID-5

К сожалению, на данным момент не удалось провести достаточное количество распределенных тестов общего блочного устройства с использованием RAID-5. RAID5 при записи производит синхронизацию избыточной информации, и это приводит к одновременному доступу с разных узлов к одной области на диске, причем этот процесс не контролируется системой разрешения конфликтов OpenGFS. По этой причине есть данные только по работе в однопользовательсокм режиме. Ниже приведены результаты для 32Gb общего устройства с SCI соединением, построенном из трех 16Gb дисков с использованием RAID-5 с избыточной информацией.

	Один тест	CPU
Запись посимвольно	5050 K/s	59%
Запись блочная	4900 K/s	28%
Перезапись	5400 K/s	30%
Чтение посимвольно	15550 K/s	76%
Чтение блочное	37050 K/s	21%

2 2004 год

2.1 Разработка программ и алгоритмов статического размещения файлов (генерация системой планирования ресурсов запросов на размещение файлов). Реализация операций копирования внутри кластеров и между ними, используя средства Globus Toolkit

2.1.1 Паспорта заданий

При решении задачи статического размещения файлов необходимо формализовать паспорт задания, а именно описание того, какие необходимы входные файлы и какие будут созданы в результате работы.

2.2 Разработка программных средств приложений, операций с группами файлов, а также возможности резервного копирования файлов для обеспечения отказоустройчивости

2.2.1 Распределённые контрольные точки

Этот механизм, известный в технической литературе как «distributed coordinated checkpointing», призван уменьшить вычислительные потери от выхода из строя нескольких вычислительных узлов. В случае сбоя необходимо пересчитывать все те задачи, которые находились на потерянном узле, контрольные точки позволяют значительно сократить затраты на восстановление задания. Различные алгоритмы рассмотрены во множестве статей, в частности [?]. В данной работе больше внимания уделялось восстановлению контрольных точек после выхода из строя узла, вопрос их создания рассмотрен не так подробно.

2.2.2 Протоколирование сообщений

Для распределённых систем передачи сообщений, например MPI, возможно в промежутках между созданиями точек сохранять на узле переданные данные. Тогда количество вычислительных ресурсов затрачиваемых на восстановление после сбоя будет значительно уменьшено, так как потребуется провести вычисления только для вышедших из строя узлов. Те процессы, которые не были подвержены сбою, не будут потреблять вычислительных ресурсов до тех пор, пока восстановленные не обработают все сообщения, переданные им с момента создания контрольной точки. Подобного рода схемы в данной работе рассмотрены не будут.

2.2.3 Краткое описание

В случае использования координированного механизма контрольных точек распределённое приложение в некоторые моменты времени сохраняет глобальное состояние задачи в дисковую память, а при сбое по этому состоянию восстанавливается рабочий процесс. В качестве простейшего алгоритма создания контрольной точки можно использовать запись всего образа процесса на диск. Используемый алгоритм характеризуется в первую очередь следующим.

- **Дополнительные затраты времени:** на сколько приложение потребует больше времени для работы по сравнению с запуском без контрольных точек.

- **Задержка на создание:** полное время, которое потребуется для создания контрольной точки. В случае, если оно меньше чем предполагаемый интервал создания новых точек, особого влияния на производительность системы не оказывается.

- **Время восстановления:** полное время, необходимое на восстановление процесса по координированному набору контрольных точек. Так как этот процесс крайне редок, скорость восстановления не играет особой роли. Отдельно стоит вопрос о миграции процессов, в данной работе не затронутый.

- **Затраченные дисковые ресурсы:** полный объем дисковой памяти, необходимой для создания одного координированного набора контрольных точек. Так как предыдущий набор не может быть удалён, пока не закончится создание следующего, в системе должно иметься место по крайней мере для двух наборов.

- **Отказоустойчивость:** количество одновременных сбоев, которые может перенести данный контрольный набор. В некоторых ситуациях разумно совмещать частую запись точек, выдерживающую сбой одного узла и вызывающую небольшое падение производительности, и редкую запись точек для восстановления после множественного сбоя.

- **Влияние на общие ресурсы:** в случае, если вычислительная среда не отдана под исключительное использование данной задаче, имеет значение тот факт, насколько система построения координированного набора контрольных точек снижает производительность всего кластера.

2.2.4 Обеспечение отказоустойчивости

Ниже, при рассмотрении различных моделей, будут использованы следующие обозначения

N — количество контрольных точек в системе.

N_{sp} — количество избыточных точек в системе.

N_{all} — общее количество узлов в системе. $N_{all} = N + N_{sp}$.

τ_r — время восстановления после сбоя. Одновременными сбоями считаются те, которые произошли на промежутке времени меньшем чем длительность процесса восстановления τ_r. В эту величину входит, в частности, время реакции системы на произошедший сбой.

Под стандартным алгоритмом ниже понимается схема, создающая только локальные контрольные точки, также известная как Course-Grained Job Swapping (CGJS). Для распределения контрольной информации по кластеру используются следующие алгоритмы.

CGJS Локальные точки. Это простейший алгоритм, не дающий никакой отказоустойчивости. В нем, на каждом узле в локальной дисковой памяти создаётся контрольная точка. Таким образом, в случае выхода из строя любого узла, информация о задаче работавшей на нем будет безвозвратно утеряна.

MIR Передача "соседним". В этом случае, копия локальной контрольной точки передаётся на "соседний" узел. Данная модель сходна с RAID1 — зеркалированием. Соседний узел определяется заранее исходя из топологии коммуникационной сети. К плюсам данной схемы следует отнести высокую скорость работы и малые вычислительные затраты. В случае коммутируемых сетей, передача данных между соседними узлами может идти независимо, что значительно снижает время на создание контрольных точек. Однако данная модель требует дискового пространства вдвое больше требуемого для CGJS и обеспечивает восстановление только при одном сбое. В случае, если выйдут из строя "соседние" узлы, одна из локальных контрольных точек будет утеряна.

CFS Передача на файловый сервер. Подходит только для систем с выделенным файловым сервером, заведомо более надёжным. Механизм обеспечения надёжности файловых серверов тщательно проработан, но выходит за рамки данной работы. В данной модели все локальные контрольные точки передаются на хранение на сервер. Такая схема обеспечивает восстановление после произвольного количества сбоев вычислительных узлов, и дисковое пространство равное требуемому для CGJS. Вероятность невосстановимого сбоя в данном случае нулевая, так как схема выдерживает сбои любого количества узлов, а сервер считается надежным.

XOR Хранение на сервере контрольной информации. В этой схеме, как и в CFS, требуется дополнительный узел, однако не обязательно более надёжный. На данном узле сохраняется побитовый XOR всех контрольных точек данного набора. Эта модель выдерживает один сбой любого узла, в частности того, на котором хранится избыточная информация. Требования к дисковому пространству — аналогичные для CGJS и дополнительно наличие места для одной точки, при N рабочих узлах требования всего в $\frac{N+1}{N}$ раз больше, чем у CGJS. В качестве преимуществ стоит отметить простоту и малые вычислительные затраты.

RSC Хранение контрольной информации на N_{sp} узлах. Этот случай имеет множество сходств с XOR, однако для вычисления избыточной информации используется не формальный XOR контрольных точек, а кодирование Рида-Соломона. Для произвольного N_{sp}, такого что $N+N_{sp} < 2^k$ и некоторого k, определяемого из используемого алгоритма кодирования, можно построить схему, устойчивую к N_{sp} сбоям и требующую в $\frac{N+N_{sp}}{N}$ раз больше дискового пространства, чем CGJS. Однако у данного алгоритма есть существенный недостаток — довольно большая вычислительная сложность, хотя в данное время этот минус не играет значительной роли. Подробнее об использованном в данной схеме методе кодирования будет рассказано ниже. Однако стоит упомянуть, что схема XOR является частным случаем данной, взятой с $N_{sp} = 1$, и алгоритмом построения избыточной информации — операцией исключающего ИЛИ (XOR).

CGJS-FS Использование стандартной схемы на общей файловой системе. В качестве альтернативы возможно использование стандартного алгоритма создания координированной системы контрольных точек, однако на общей кластерной файловой системе. Тогда задача защиты от сбоев снимается со схемы создания контрольных точек. Количество сбоев, которое выдержит данная система, и скорость её работы зависят исключительно от модели, использованной в общей кластерной файловой системе. В качестве положительного момента такой схемы стоит отметить её простоту и малые непосредственные вычислительные затраты. Конечно, аппарат файловой системы потребует некоторых ресурсов — дисковых, процессорных и сетевых для обеспечения хранения и защиты этих данных от сбоя, но это уже косвенные издержки. В зависимости от модели файловой системы данная схема близка либо CFS, XOR или RSC. Однако если в случае кластерной файловой системе на общем дисковом накопителе особой разницы с CFS нет, то в случае вычислительных кластеров с распределёнными локальными дисковыми накопителями сильно возрастает время восстановления по сравнению со схемами XOR и RSC.

2.2.5 Многоуровневые схемы

Для большинства распределённых систем одиночные сбои происходят гораздо чаще, чем многократные. Соответственно, использование схемы, устойчивой к k сбоям может создать неоправданно высокие затраты. Многоуровневые схемы представляют из себя комбинацию простой модели типа MIR или XOR и сложной высоко-затратной типа CFS или RSC. Тогда наиболее вероятные одиночные сбои будут восстанавливаться за меньшее время, чем сбои множественные. В частности, в [?] показано, что двухуровневые модели в среднем дают меньшие дополнительные затраты, нежели схемы одноуровневые. Так в случае простых схем, выдерживающих один сбой, после выхода из строя второго узла вычисления должны начаться заново, а сложные схемы дают слишком большую прибавку времени. Многоуровневые схемы характеризуются, в частности, соотношением количества N_1 и N_2 контрольных точек 1 и 2 уровней.

При росте N_1/N_2 слишком велико, время восстановления после двойного сбоя также растёт. Однако в случае, если это соотношение мало, значительными становятся дополнительные временные затраты по сравнением с системой без контрольных точек. Соотношение тяжёлых и лёгких точек, ровно как и максимальное количество сбоев, выдерживаемых схемой, должно быть установлено исходя из имеющейся вычислительной системы и требований на её отказоустойчивость.

2.2.6 Многоуровневая схема RSC_k

Данная схема позволяет сделать многоуровневую схему в виде одной модели создания избыточной информации всего лишь модификацией параметров. Обозначим RSC_k схему RSC при $N_{sp} = k$. Тогда RSC_1 является "лёгкой" схемой с малым временем восстановления и устойчивой к одному сбою. Но переход от RSC_k к RSC_{k+1} — это добавление одной новой точки избыточной информации, без модификации имеющихся k. Это же верно для большего шага, то есть для перехода от RSC_k к $\mathsf{RSC}_{k+\kappa}$, отличие в том, что создаётся не одна дополнительная точка избыточной информации а κ. Таким образом, создание RSC_k может быть разбито на несколько этапов постепенно повышающих отказоустойчивость системы. Так, быстро созданная RSC_1 позволит отработать наиболее частый одиночный сбой, даже если тот произошёл в процессе создания "тяжёлой" точки. Данная модель может иметь как два так и много уровней. Так, для $k = 3$ возможны два варианта:

2 уровня, при использовании схем RSC_1 и RSC_3,

3 уровня, при использовании схем RSC_1, RSC_2 и RSC_3.

2.2.7 RSC с модифицированной матрицей

2.3 Оценка затрат ресурсов при обслуживании запросов на размещение файлов (время выполнения, дополнительные дисковые и коммуникационные затраты). Оценка эффективности оптимизации статического размещения файлов, создания копий.

2.3.1 Ресурсам требуемые различными схемами контрольных точек

В следующей таблице кратко описаны требуемы ресурся для рассмотренных выше схем контрольных точек, как дисковые так и сетевые. Кроме этого указана отказоустойчивость схем.

Схема	Дисковые ресурсы	Сетевые ресурсы	Отказоустойчивость
CGJS	N локальных контрольных точек	не требует	не обеспечивает
MIR	CGJS + одна копия каждой на удалённом узле	одновременная передача N точек на разные узлы	устойчива к одиночному сбою, в некоторых случаях к нескольким
CFS	N контрольных точек на центральном сервере	передача всех точек на центральных сервер	любое количество сбоев, кроме серверного
XOR	CGJS + одна точка избыточной информации	передача объёма равного N точкам по частично независимым каналам	один сбой, как узла содержащего контрольную точку, так и избыточную
RSC	CGJS + N_{sp} точек избыточной информации	передача объёма равного $N \cdot N_{sp}$ точкам по частично независимым каналам	N_{sp} сбоев, как узлов содержащих контрольные точки, так и избыточные

2.3.2 Оптимизация широковещательных рассылок в однородных средах.

Ниже будут рассмотрены некоторые методы распределения данных по однородному вычислительному кластеру и проанализирована их производительность.

2.3.3 Постановка и моделирование

Требуется передать файл с исходного узла на n принимающих, размер файла может быть весьма велик и доходить до десятков гигабайт. Передача может вестись не только с исходной узла ($source$), возможна ретрансляция узлами-приёмниками. Таким образом, строится некоторый ориентированный граф, охватывающий все узлы-приёмники. Ниже будет представлено несколько различных типов передающих сетей. Общие для всех моделей обозначения:

N_{dest} — количество принимающих узлов.

B_{disk} — скорость чтения единицы данных из локальной дисковой памяти.

L_{disk} — задержка перед чтением любого объёма данных из локальной дисковой памяти.

B_{net} — скорость чтения единицы данных по сетевому соединению.

L_{net} — задержка перед чтением любого объёма данных по сетевому соединению.

B_{total} — скорость передачи данных в построенной коммуникационной сети. Можно взять равной средней скорости выхода данных с исходного узла по каждому соединению.

Для простейшей ситуации, когда $N_{dest} = 1$, скорость передачи можно считать равной $B_{total} = \min(B_{disk}, B_{net})$.

2.3.4 Передача от одного всем

Простейшая модель, когда исходный узел передаёт каждому принимающему непосредственно, без ретрансляций (рис. 1). Эта схема отличается простотой, однако для неоднородных коммуникационных сетей она не применима. Её отличает простота реализации, однако в случае, если $B_{net} < B_{disk} \cdot N_{dest}$, начнутся значительные потери производительности. Скорость B_{total} в такой ситуации можно считать равной $B_{total} = \min(B_{net}/N_{dest}, B_{disk})$.

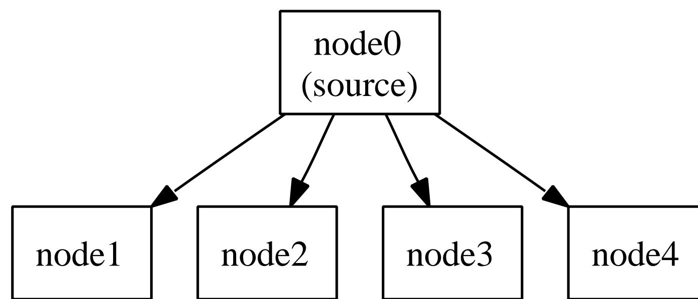

Рис. 1: Передача от одного ко всем. $N_{dest} = 4$

2.3.5 Последовательная передача

В этой модели блоки данных передаются по цепочке узлов (рис. 2). Каждый ретранслирующий узел записывает полученный блок на локальный диск и передаёт следующему в цепочке. Эта схема значительно разгружает сетевую инфраструктуру, так как ни один узел не имеет более одного входящего и одного исходящего потока. Таким образом, на сетях с коммутируемыми каналами данные передачи будут идти практически независимо. Кроме этого, в случае гетерогенной коммуникационной сети возможно так расположить узлы в последовательность, что по наиболее узким местам произойдёт только одна передача. Время, затраченное на передачу блока равно времени затраченному на запись блока на диск и пересылку по сети. Таким образом, для линейной передачи пропускная способность $B_{total} = \frac{B_{net} \cdot B_{disk}}{B_{net} + B_{disk}}$.

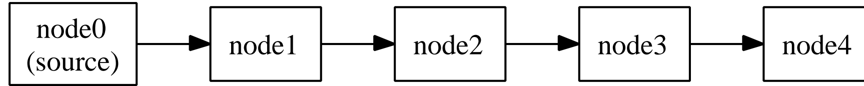

Рис. 2: Последовательная передача. $N_{dest} = 4$

2.3.6 Передача по двоичному дереву

Эта модель построена аналогично последовательной передаче, однако некоторые узлы дополнительно передают блоки данных двум другим (рис. 3). Занумеруем узлы каким-либо образом, установив исходному узлу индекс 0. Тогда по данным индексам построим двоичное дерево с корнем в узле с нулевым индексом. Построение происходит по двоичной записи индекса — у $node_k$ источником является $node_{k_R}$, а передача идёт на $node_{k_L}$ и $node_{k_L|1}$, где k_R — k побитовый сдвиг вправо, а k_L — влево. Таким образом, для любого N_{dest} мы построим дерево глубины не более $[\log N_{dest}] + 1$. Вычисление скорости передачи аналогично последовательному случаю $B_{total} = \frac{B_{net} \cdot B_{disk}}{B_{net} + B_{disk}}$. Также, данная схема может быть адаптирована под гетерогенные коммуникационные сети, если при нумерации узлов учитывать информацию о топологии вычислительной системы.

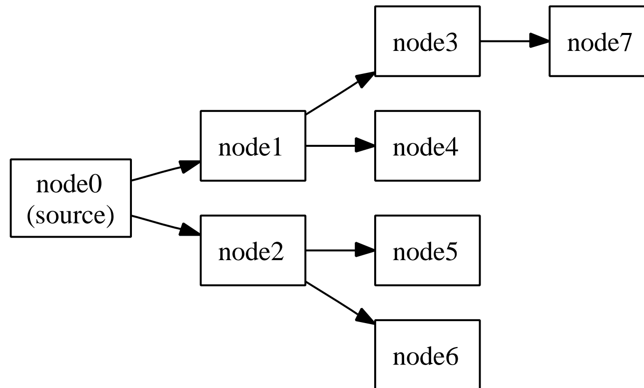

Рис. 3: Передача по двоичному дереву. $N_{dest} = 7$

2.3.7 Сравнительные результаты

В ходе тестов передача данных шла с использованием коммуникационной среды SCI с пропускной способностью около 240Mb/s и жёсткие диски со скоростью около 40Mb/s. Передача данных по сетевой среде производилась посредством средств MPI, что объясняет некоторые странности в полученных результатах. Для модели передачи по дереву были достигнуты скорости $B_{total} = 34.5\text{Mb/s}$ при $N_{dest} = 7$. При аналогичных условиях для последовательной пересылки и для передачи от одного всем были получены скорости $B_{total} = 28.5\text{Mb/s}$ и $B_{total} = 19.5\text{Mb/s}$ соответственно. На графике (рис.

4) представлены усреднённые результаты тестов на файлах размером от 2 до 16 Gb по трём рассмотренным выше моделям коммуникационных сетей.

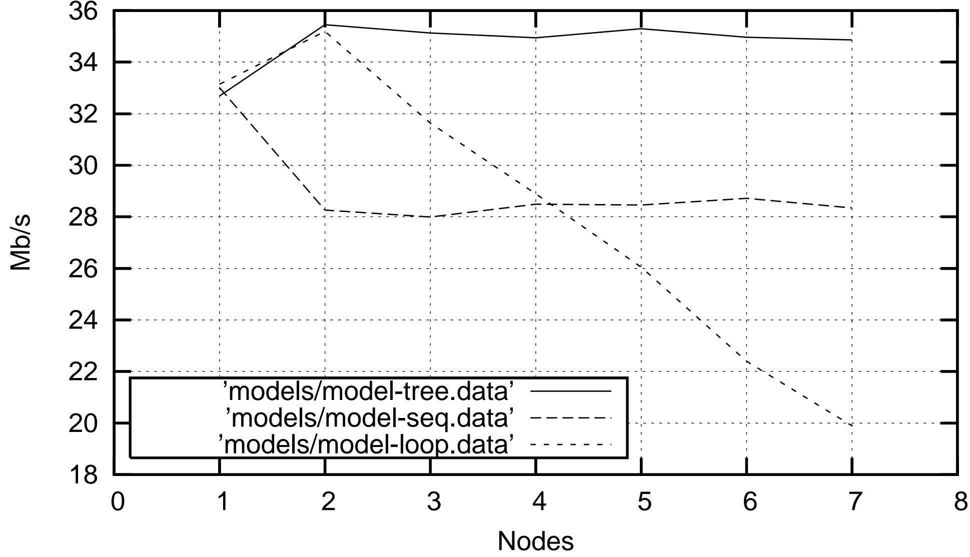

Рис. 4: Скорость передачи B_{total} в зависимости от количества узлов. $N_{dest} = 1 - 7$

2.4 Оптимизация широковещательных рассылок в неоднородных средах.

2.4.1 Постановка задачи.

Рассмотренные выше модели распределения данных подходят только для однородных коммуникационных сред, так как не учитывают сложность передачи между компонентами связности метакластера. Ниже будет предложен простой алгоритм построения графа пересылок и доказательство его оптимальности.

2.4.2 Формализация задачи.

Дан полный граф Γ с вершинами $n(\Gamma) = \{n_0, \dots, n_N\}$, вершина n_0 и весовая функция на ребрах $f : n(\Gamma) \times n(\Gamma) \to \mathbb{R}$. Для ребра $e = (n_i, n_j)$ положим $f(e) = f(n_i, n_j)$. Далее везде f предполагается симметричной, то есть $f(x, y) = f(y, x)\ \forall x, y \in n(\Gamma)$.

Определение 1. Весом графа Γ' с ребрами $e(\Gamma') = \{e_1, \dots, e_N\}$ называется функция равная $\mu(\Gamma') = \sum_{1 \dots N} f(e_i)$.

Определение 2. Граф Γ' называется связным, если для любого $n_i \in n(\Gamma)$ существует путь по ребрам $e_1^i, \dots, e_k^i$ из n_0 в n_i

Определение 3. Связный граф Γ' называется оптимальным, если его вес минимален среди всех связных графов Γ'' с теми же вершинами.

Требуется построить оптимальный граф Γ' с вершинами $n(\Gamma') = n(\Gamma) = \{n_0, \dots, n_N\}$. Заметим, что оптимальных графов может быть много, например в случае если $\forall i, j\ f(n_i, n_j) = \text{const}$, оптимальным будет любой связный граф.

2.4.3 Построение.

Будем строить Γ' по индукции. База: $\Gamma'_0 = (n'_0, \varnothing)$ — единственный граф из одной вершины и поэтому оптимальный.

Шаг: пусть имеется оптимальный граф Γ'_k , построим Γ'_{k+1} с теми же свойствами. Пусть $n(\Gamma'_k) = \{n'_0 = n_0, n'_1, \dots, n'_k\}$ — множество вершин Γ'_k, найдем вершины $n'_{k+1} \in n(\Gamma) \setminus n(\Gamma'_k)$ и $n'_j \in n(\Gamma'_k)$ такие, что $\forall \hat{n}' \in n(\Gamma'_k), \forall \hat{n} \in n(\Gamma) \setminus n(\Gamma'_k)\ f(n'_j, n'_{k+1}) \leqslant f(\hat{n}', \hat{n})$. То есть, n'_{k+1} — вершина из $\Gamma \setminus \Gamma'_k$, такая что "расстояние" от нее до Γ'_k минимально среди всех $n(\Gamma) \setminus n(\Gamma'_k)$

Будем нумеровать ребра в порядке добавления в Γ'_{k+1}, то есть, $\Gamma'_{m+1} = \Gamma'_m \cup \{e_m\} \cup \{n'_m\}$.

Утверждение. Γ'_{k+1} *оптимален.*

Рассмотрим произвольный связный граф без циклов $\Gamma'' = \Gamma''_{k+1}$ на вершинах $n(\Gamma'_{k+1})$. Покажем что $\mu(\Gamma'_{k+1}) \leqslant \mu(\Gamma'')$.

Определение. Положим характеристику $\chi_{n'_l}(\hat{\Gamma})$ равной количеству ребер в графе $\hat{\Gamma}$ имеющих одним из концов вершину n'_l.

Пусть $n'_{l_1}, n'_{l_2}, \dots, n'_{l_q},\ q = \chi_{n'_{k+1}}(\Gamma'')$ — вершины, соединенные с n'_{k+1} ребром в графе Γ'', выписаные в порядке добавления их в граф Γ'_{k+1}, то есть, $\forall j \in \{1, \dots, q-1\}\ \exists i\ n'_{l_j} \in n(\Gamma'_i),\ n'_{l_{j+1}} \notin n(\Gamma'_i)$. Далее рассмотрим два случая.

$\chi_{n'_{k+1}}(\Gamma'') = 1$. Тогда $f(n'_{l_1}, n'_{k+1}) \geqslant f(n'_j, n'_{k+1})$, где n'_j — вершина, соединенная с n'_{k+1} ребром в графе Γ'_{k+1}, а граф Γ'_k оптимален по предположению индукции.

$\chi_{n'_{k+1}}(\Gamma'') > 1$. Построим граф $\hat{\Gamma}''$ с весом $\mu(\hat{\Gamma}'') \leqslant \mu(\Gamma'')$ такой, что $\chi_{n'_{k+1}}(\hat{\Gamma}'') = \chi_{n'_{k+1}}(\Gamma'') - 1$ Положим $i_1 = l_q$ и $i_2 = l_{q-1}$. Рассмотрим ребро $\hat{e}_2 = (n'_{k+1}, n'_{i_2})$. Так как Γ'' не имеет циклов, после удаления $\hat{e}_2$ он распадется на две компоненты связности. Обозначим Γ''_+ компоненту содержащую n'_{i_2} и Γ''_- — оставшуюся часть.

Утверждение 1. Для любой вершины $n'_l \in n(\Gamma'_{k+1})$, ребра $e_m \in e(\Gamma'_{k+1})$, выходящего из нее, и любого ребра $e_i \in e(\Gamma'_{k+1}),\ l < i \leqslant m$ верно, что $f(e_i) \leqslant f(e_m)$.

Утверждение 2. Для любой вершины $n'_l \in n(\Gamma'_{k+1})$, ребра $e \notin e(\Gamma'_{k+1})$ или $e = e_{k+1}$, выходящего из нее, и любого ребра $e_i \in e(\Gamma'_{k+1}),\ l < i \leqslant k+1$ верно, что $f(e_i) \leqslant f(e)$.

Определение. Назовем ребро $e = (n_i, n_k)$ *соединяющим*, если $e \in e(\Gamma'_k)$ и $n_i \in n(\Gamma''_+),\ n_j \in n(\Gamma''_-)$ или $n_j \in n(\Gamma''_+),\ n_i \in n(\Gamma''_-)$.

Предположим что найдется соединяющее ребро $e_{j'_2}$, такое что $i_2 < j'_2 \leqslant i_1$, тогда, $f(e_{j'_2}) \leqslant f(e)$ по утверждению 2. Рассмотрим граф $\hat{\Gamma}'' = e_{j'_2} \cup \Gamma'' \setminus e$, для него верно, что $\mu(\hat{\Gamma}'') \leqslant \mu(\Gamma'')$ и $\chi_{n'_{k+1}}(\hat{\Gamma}'') = \chi_{n'_{k+1}}(\Gamma'') - 1$, таким образом построен граф с искомыми свойствами. Иначе, рассмотрим вершину $n'_{i_3} \in \Gamma'_-$ такую, что из нее выходит ребро e_{j_3}, $j_3 > i_2$. Тогда, $f(e_{j_3}) \leqslant f(e)$ по утверждению 1. Если найдется соединяющее ребро $e_{j'_3}$, такое что, $i_3 < j'_3 \leqslant i_2$ то заменой e на $e_{j'_3}$ получаем граф $\hat{\Gamma}''$ с искомыми свойствами, иначе — найдем $n'_{i_4} \in \Gamma''_+$ и проведем с ней аналогичные рассуждения. В результате получится цепочка неравенств $f(e) \geqslant f(e_{j_3}) \geqslant f(e_{j_4}) \geqslant \dots$. Поскольку j_m ограничены снизу и существует хотя бы одно соединяющее ребро, таковое будет найдено и заменой e на него будет построен искомый граф $\hat{\Gamma}''$, такой что $\mu(\hat{\Gamma}'') \leqslant \mu(\Gamma'')$ и $\chi_{n'_{k+1}}(\hat{\Gamma}'') = \chi_{n'_{k+1}}(\Gamma'') - 1$.

Таким образом цепочкой переходов $\Gamma'' \to \hat{\Gamma}'' \to \dots$ можно придти к случаю $\tilde{\Gamma}''$, где $\chi_{n'_{k+1}}(\tilde{\Gamma}'') = 1$.

2.4.4 Заключение.

Предложенный алогритм может быть использован для частичной оптимизации схем широковещательных рассылок в метакластерах. В описанном методе никак не учитываются более или менее эффективные способы передачи данных в однородной коммуникационной среде. Однако, при непосредственной реализации алгоритма возможно учитывать различные подходы к оптимизации гомогенных широковещательных рассылок. Также не рассмотрена возможность параллельного исполнения некоторых операций, эта задача должна быть возложена на иные средства.

2.4.5 Метакластера

Задачи оптимзиации для метаклестера можно свести к рассмотрению с дополнительным условием при выборе ребра. В указанном алгоритме не наложено никаких ограничений на выбор ребра в случае, если есть несколько с равным весом. Положим, что в такой ситуации выбирается наиболее близкое к корню. Для простоты рассмотрим модель метакластера, состоящего из двух тесносвязанных компонент соединенных между собой медленным каналом. Пусть метрика, то есть вес ребра, внутри каждого кластера - 1, а между компонентами - 5. Тогда, начиная с одно из узлов кластера будет построен граф (рис. 5), имеющий несколько ребер с одинаковым весом выходящих из одной вершины.

Если предположить что компоненты связности метакластера однородны, то для каждой такой группы ребер возможно применить оптимизации рассмотренные в разделе про однородные широковещательные пересылки. После изменения схемы широковещательной рассылки на двоичное дерево, результирующий граф будет выглядить иначе (рис. 6). При этом увеличивается глубина графа, однако при большом числе принимающих узлов внутри одной однородной компоненты это может быть эффективно. Каждый конкретный случай требует тщательного анализа, а так же зависит от текущей загрузки магистрального канал между компонентами.

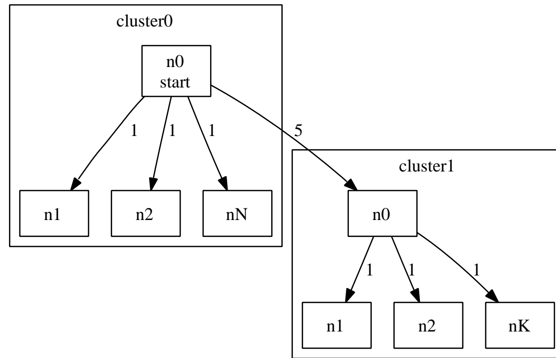

Рис. 5: Граф передач для неоднородного кластера

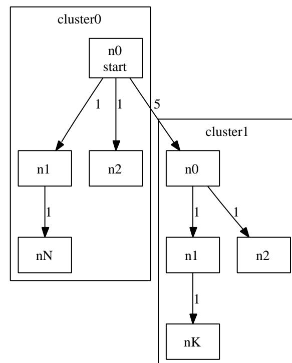

Рис. 6: Оптимизированный граф передач для неоднородного кластера

3 2005 год

3.1 Файловый индекс

3.1.1 Общее описание

Система предназначена для сбора, хранения и анализа информации о состоянии локальных дисков параллельной вычислительной системы с целью обеспечить возможность работы с группами файлов, представляющих состояния вычислительных процессов.

Основными сущностями в системе управления ЛДП являются: диск — локальная файловая система некоторого вычислительного узла, файл, задача и контрольная точка — группа файлов, ассоциированная с задачей. Создаваемые по мере работы приложения контрольные точки регистрируются в системе, после чего происходит планирование операций создания избыточной информации для предотвращения возможных сбоев.

Благодаря тому, что вся информация находится в файловом индексе, можно обеспечить выполнение операций восстановления в случае сбоев или временной недоступности вычислительных узлов, проверку целостности файлов и их групп, а также автоматическое удаление устаревшей информации.

Основной менеджер системы реализован с использованием языка высокого уровня Haskell, что позволяет обеспечить удобное изменение алгоритмов поддержания целостности и легкость построения и проверки математических моделей.

3.1.2 Интерфейс

Процесс создает сетевой сервер на TCP порту 2563 и считывает команды из стандартного ввода. Ниже приводятся стандртые команды с описанием их действия и разбиению по группам.

Контрольные команды. Команды, используемые для управлением работой файлового индекса, такие как отключение сетевого сервера, отключение клиента или загрузки другой базы.

- halt Остановить сервер без сохранения базы.
- shutdown Остановить сервер, предварительно сохранив базу в файл save.cpidb
- start Запустить сетевой сервер. Имеет смысл лишь при запуске индекса в безсетевом режиме.
- stop Остановить сетевой сервер. Считывание из стадартного ввода будет производится в упрощенном режиме.
- save Выдать в поток список команд для создания базы в ее текущем состоянии. Поток после этого закрывается.

- load:test Загрузить тестовую базу.
- /cntl/load «имя файла» Загрузить базу из указаного файла
- /cntl/save «имя файла» Сохранить текущую базу в указаный файл
- /cntl/close Закрыть текущее соединение. Используется для закрытия файлов, запись в которые завершена. Игнорируется для стандартного ввода.
- /cntl/shutdown Выставить флаг остановки, при закрытии файла будет послана команда halt. Используется командой shutdown

Команды по добавлению и удалению информации. Используются, в частности, прикладной библиотекой libccp2 для оповещения о создании новой контрольной точки.

- /cp/add «uuid точки» «uuid задания» «схема» [«uuid файла» «номер в наборе»] Добавить контрольную точку с данными параметрами. В случае, если таковая уже есть, информация в ней будет обновлена, а списки файлов конкатенированы. Если какое-то из полей равно «-» то будет использвано аналогичное из старой. Это позволяет обновлять лишь ту информацию, которая изменилась. Пара значений в квадратных скобках может повторяться столько раз, сколько необходимо в пределах размера принимающего буфера.
- /cp/remove «uuid точки» Удалить контрольную точку с указаным uuid-ом из индекса. Никаких действий по удалению данных из файловой системы не производится.
- /file/add «uuid файла» «uuid диска» «имя файла» «контрольная сумма файла» [«uuid точки» «номер в наборе»] Добавить информацию о файле. В случае, если таковой уже есть, то есть существует запись с таким-же uuid-ом, инфрмация о нем будет обновлена согласно схеме описаной для добавления контрольной точки.
- /file/remove «uuid файла» Удалить файл с указаным uuid-ом из индекса. Выполняется комада удаления его из файловой системы.
- /disk/add «uuid диска» «узел» «точка монтирования» Создать новый диск с указаными параметрами. В случае, если диск с таким uuid уже существует, вся, в отличии от фалойв и контрольных точек, информация будет обновлена.
- /disk/remove «uuid диска» Удалить указаный диск из индекса.
- /task/remove «uuid задачи» Удалить все контрольные точки принадлежащие данной задаче.

Команды приведения индекса к целостному виду.

- rebuild Проверить и пересоздать списки ссылок на файлы в контрольных точках и в файлах на точки.

- methods Привести обозначения схем обеспечения отказоустойчивости в контрольных точках к каноническому виду, то есть к виду «схема»-«число файлов»[-«избыточность»], неизвестные схемы будут заменены на plain, в случае если избыточность нулевая, то схема будет иметь вид «схема»-«число файлов»

- consistent Удалить из базы ненужную информацию. Будут удалена следующая информация:

  - файлы, ссылающиеся на несуществующий диск.
  - файлы, которые не принадлежат ни одной контрольной точке
  - точки, не содержащие ни один файл
  - точки, для которых невозможно восстановление

На данный момент не реализована система разграничения прав в связи с тем, что система еще не подошла к стадии внедрения и простота отладки ставится выше необходимости оградить систему от несанкционированного доступа.

3.1.3 Структура данных

Диск описывается следующими полями:

- уникальным идентификатором (uuid),

- узлом (ip адрес или доменное имя, любая информация по которой ssh сможет установить соединение)

- точкой монтирования.

В случае удаления диска, все файлы на него ссылавшиеся считаются несуществующими и удалятся при следующем приведении базы к целостному виду.

Информация, описывающая файл:

- уникальный идентификатор,

- идентификатор диска, на котором он расположен

- имя файла в файловой системе

- контрольная сумма файла, необходима для поиска поврежденных фйлов

- список ссылок на контрольные точки, которым он принадлежит в формате (идентификатор точки, номер в наборе)

Для контрольной точки определены следующие поля

- уникальный идентификатор,
- идентификатор задачи, которой принадлежит данная точка
- имя схемы обеспечения отказоустойчивости в формате, описаном выше
- список ссылок на файлы, которые ей принадлежит в формате (идентификатор файла, номер в наборе). Двойная запись позволяет восстанавливать потеряные ссылки

Так же, неявно сщуствует список известных схем обеспечения отказоустойчивости. Они описываются функцией, которая по набору номеров файлов в наборе дает ответ, является ли точка невосстановимой и можно ли ее удалить из системы.

3.1.4 Библиотека оповещения о создании контрольной точки libccp2

Данная библиотека позволяет вычислительному приложению оповещать систему обеспечения отказоустойчивости о создании координированной контрольной точки. При этом на файловый индекс cpi TCP/IP передается вся необходимая информация о координированной контрольной точке. libccp2 является переработкой описаной в предыдущем отчете прикладной библиотеки и практически полностью совместима с ней по API. В осовном изменения коснулись протокола передачи данных демону, двоичный протокол именен на текстовый, и добавлены поля UUID для контрольной точки и отдельных файлов. К сожалению, пока доступен лишь функционал работы с координированными наборами.

Оповещение сервера происходит следующим образом. Устанавливается TCP/IP соединение с фаловым индексом и на него отправляются команды создания контрольной точки и набора ее файлов. При этом передается

- uuid точки, создается при отправке, подробнее в описании

  ```
  CCPC_FLAGS_UUID_GEN
  ```

- uuid задачи, считывается из переменной окружения TASK_UUID при старте вычислительного процесса.
- схема обеспечения отказоустойчивости выставляется в plain, то есть без избыточной информации.

Далее для каждого файла передается

- абсолютное имя файла
- uuid узла или файловой системы, по которому можно определить расположение файла

- последовательный номер файла контрольной точки
- идентификатор контрольной точки, для связи в индексе

3.1.5 Пользовательский интерфейс

Пользовательский интерфейс библиотеки состоит из вызовов инициализации, непосредственно передачи сообщений и вызова завершения работы. Ниже они описаны более подробно

`ccpc_flags_set(int flags)`

Устанавливает различные флаги, так или иначе изменяющие поведение библиотечных вызовов. В качестве параметра возможно логическое ИЛИ следующих флагов:

`CCPC_FLAGS_DEBUG`

Активировать дополнительные отладочные распечатки о посылаемой на сервер информации.

`CCPC_FLAGS_NODENAME`

Брать в качестве UUID файловой системы первые 16 символов имени узла, получаемого через uname

`CCPC_FLAGS_ENVUUID`

Брать в качестве UUID приложения значение из переменной окружения с именем TASK_UUID. В случае если она содержит некорректно отформатированное значение будет выдано сообщение об ошибке и идентификатор приложения будет взят нулевым.

`CCPC_FLAGS_SEQ_ID`

В случае, если выставлен этот флаг, на сервер будут отсылаться не имена файлов, а номера контрольных точек для соответствующих узлов. Имя файла будет создано только для локальной точки. Подробнее об этом в описании функции ccpc_register. В данный момент в libccp2 этот функционал неработоспособен.

`CCPC_FLAGS_UUID_GEN`

Создавать уникальные идентификаторы файлов и контрольных точек при их отправке. Считываются из /proc/sys/kernel/random/uuid. В будущем, возможно, будет заменено на использование вызовов из системной библиотеки libuuid

`ccpc_flags_unset(int flags)`

Аналогично функции ccpc_flags_set, только снимает флаги, указанные в качестве аргумента.

`ccpc_init(int numproc, int templ_size, const char * templ)`

Заполняет внутренние структуры информацией о файловой системе и идентификаторе приложения, а также инициализирует шаблоны имен файлов контрольных точек. Шаблон может содержать две подстановки целочисленных переменных в формате описанном для библиотечной функции printf. Первая будет заменена на номер узла для которого создается файл контрольной точки, вторая — на номер точки. Пример: "e2d.%02d-%04d.ccp". Данный вызов должен быть произведен до любого применения ccpc_register. Перед ним возможно вызывать только ccpc_flags_set или ccpc_flags_unset

`ccpc_init_serv(const char * serv_addr)`

Устанавливает IP адрес сервера, получающего сообщения. Возможно любое корректное значение в стандартном формате.

`ccpc_uuid_get_fs ( char * uuid )`

Возвращает значение полученное для UUID локальной файловой системы. В данный момент это либо нули, либо первые 16 символов имени узла полученного через вызов uname. Это зависит от того, выставлен ли флаг CCPC_FLAGS_NODENAME. Аргумент должен ссылаться на область памяти больше 16 байт, значение UUID будет возвращено в двоичном виде а не в печатном формате.

`ccpc_uuid_set( int rank, const char * uuid )`

Устанавливает значение UUID файловой системы для узла с идентификатором rank. Используется впоследствии при заполнении данных о файле контрольной точке, не используется при установленном флаге CCPC_FLAGS_SEQ_ID. Значение uuid — 16 байт двоичной информации, а не UUID в стандартном формате. Может быть использован с вызовом ccpc_uuid_get_fs для обмена информацией о UUID файловых систем на различных узлах.

`ccpc_rank_local(int rank)`

Устанавливает идентификатор локального узла, используется только при выставленном флаге CCPC_FLAGS_SEQ_ID.

`ccpc_register(const int * seq_id, int size)`

Отправить на сервер информацию о наборе файлов для контрольной точки. В параметре seq_id передается массив последовательных номеров файлов локальных точек на момент, когда был создан файл на данном узле. При этом будут произведены подстановки в шаблон, заданный вызову ccpc_init и полученные имена отправлены на сервер. В случае, если выставлен флаг CCPC_FLAGS_SEQ_ID, подстановка будет проведена только для локального узла, номер которого должен быть выставлен вызовом ccpc_rank_local. Для остальных будет отправлено не имя файла, а только номер точки.

`ccpc_terminate()`

Отсылает на сервер данные о последней точке, прошедшей вызов ccpc_register. При этом, устанавливаются флаги, указывающие о том, что это последнее сообщение от данного процесса. После этого не должно быть вызовов ccpc_register, только ccpc_fini. Данная функция имеет смысл лишь при отсутствии флага CCPC_FLAGS_SEQ_ID и если вся информация отсылается одним выделенным процессом задания. В libccp2 данный вызов оставлен для обеспечения обартной совместимости, однако выполняемые им действия особого смысла не имеют.

`ccpc_fini()`

Освобождает все ресурсы, занятые вызовами данной библиотеки.