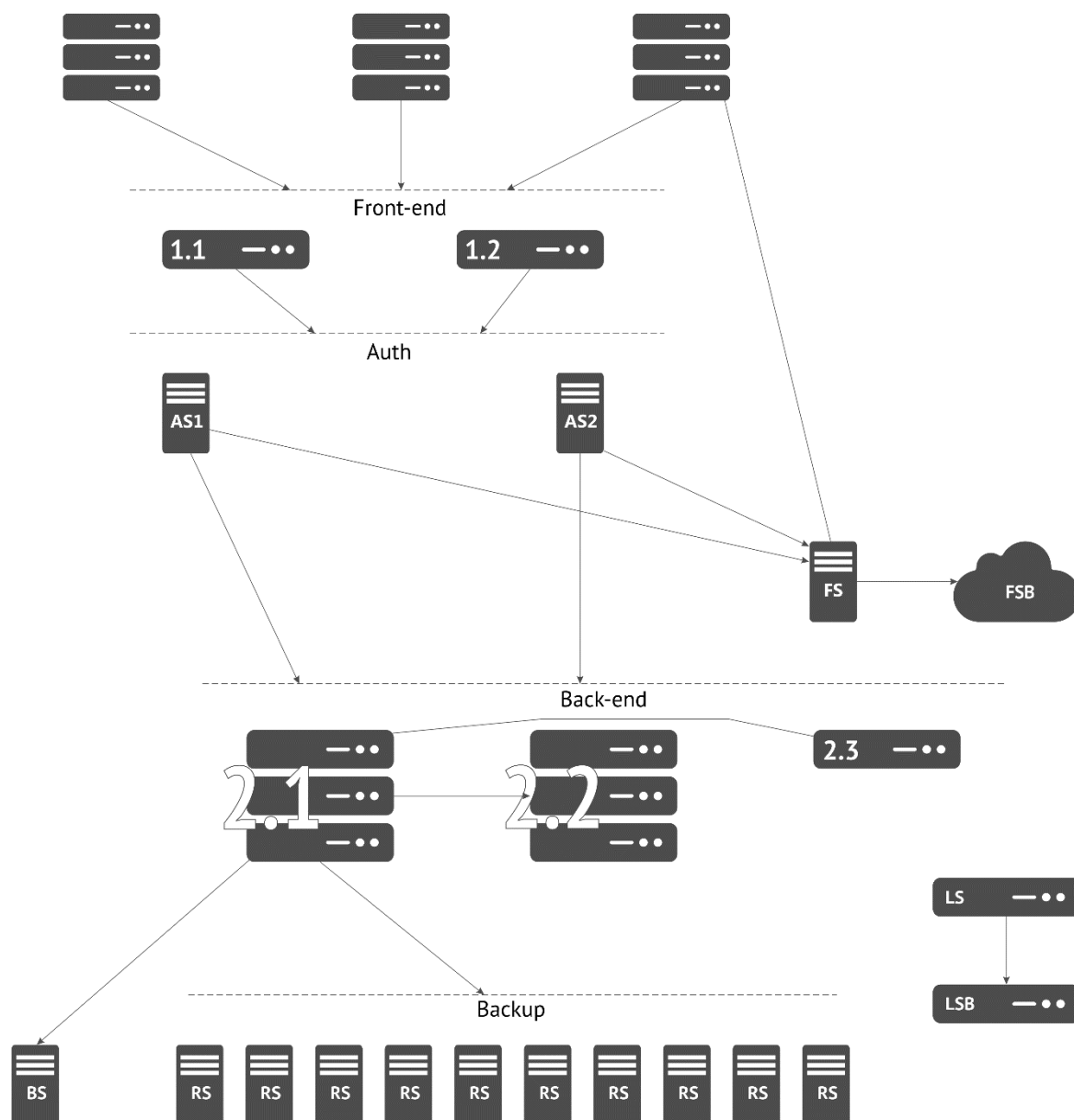


Архитектура кластера хранения данных

В целом система была задумана как кластер для безопасного хранения конфиденциальных и не очень конфиденциальных данных достаточно разнородных интернет-ресурсов и мобильных приложений с быстрым доступом. Некоторые части архитектуры, которые я специально выделяю ниже, могут быть весьма странными и неоднозначными, однако, я вас уверяю, это требования заказчика.

Схема 1. Общее представление архитектуры кластера



Суть

Зоопарк вышеописанных проектов (сайты, мобильные приложения) будет обращаться в нашему хранилищу для записи и чтения данных.

Текстовая информация должна преимущественно храниться в базах данных PostgreSQL. Однако, для некоторых видов информации предусмотрены колоночные и key-value хранилища, но задачи для подобных альтернативных решений пока определены не точно, поэтому на первом этапе разработки они не будут внесены в архитектуру.

Файлы будут храниться в файловом хранилище представляющим собой обычный каталог с подкаталогами, в которых будут храниться файлы и которые будут бэкапиться в хранилище Google Drive. Существует так же альтернативный подход к хранению, заключающийся в написании собственного софта аналогичного Haystack от Facebook, но решение о таком развитии архитектуры может быть принято только после старта первой версии системы. Попробуем, посмотрим...

О хранении файлов будет отдельная часть позже.

Большая часть запросов, приходящих на вход кластера будет представлять собой HTTP-пакеты, на борту которых будет закодированное сообщение о том что же мы все-таки хотим сделать с данными, чаще всего запрос или несколько запросов к базам данных. Эти сообщения будут попадать на уровень front-end (сервера 1.1 и 1.2 на схеме, далее FES).

Кроме того, будут HTTP-запросы типа POST или PUT, которые будут нести с собой файлы для записи в файловое хранилище, а так же метаданные, о которых подробнее будет рассказано в разделе о хранении файлов. В этом случае, если запись файла будет разрешена он попадет в хранилище, а затем будет среплицирован в резервное хранилище в облаке Google.

Но не HTTP единым...

Как же WebSocket и прочие UDP?

У нас их не будет. Пока не будет. Пока в них нет необходимости, широкого применения им не найти, а частные случаи использования лучше решать проектам-клиентам «на местах».

Серверами исполняющими запросы являются узлы уровня back-end, в основном это машины 2.1 и 2.2; машина 2.3 так же участвует в исполнении запросов, на ней исполняются только особо важные запросы, такие как платежи и переводы денежных средств, изменение основных данных пользователей и т. п. Таким образом 2.3 есть вспомогательный сервер для обеспечения большей надежности и безопасности хранения критичных данных. В общем и целом, для особо важных данных будет синхронная репликация на 2.2 с подтверждением записи на диск и отдельное сохранение на 2.3, все в одной транзакции..

Собственно, большая часть работы кластера будет крутиться вокруг исполнения запросов к базам данных, лежащих на серверах уровня back-end. Ноды 2.1 и 2.2 связаны как master-slave сервера и в лучшем случае дублируют друг друга. Предполагается что 2.1 и 2.2 будут работать совместно: запросы на чтение будут

выполняться на обоих серверах (балансировка на чтение round-robin), а запросы на изменение данных – только на 2.1 .

Далее предполагается что полезные данные будут «растекаться» по машинам уровня backup, которые могут быть распределены географически. Решение может выглядеть весьма странным: зачем плодить столько отдельных бэкап-серверов, но причина выбора такой конфигурации этого уровня выбрано заказчиком, скорее, из политических соображений, нежели из объективной необходимости – каждый проект, использующий нашу систему хранения хочет хранить копию своих данных у себя на серверах (если таковые имеются) и взаимодействовать с этой копией.

Не обойдется, безусловно, и без бэкап-сервера, объединяющего в себе всю хранящуюся информацию из базы данных (узел BS).

Кроме того, в систему будет включена пара серверов хранения логов (машины LS и LSB). По названиям, думаю, должно быть понятно, что главный здесь сервер LS, а LSB является бэкапом для LS. Наличие LSB нельзя назвать обязательным, однако в случае его отсутствия желательно иметь RAID-массив на LS, например, RAID 10, с батареей, ибо потеря логов хоть и не критична, но может быть весьма неприятна.

Отдельно стоит остановиться на сервере авторизации (AS) и, собственно, системе авторизации и прав на исполнение запросов. Требования к безопасности данных привели к тому, что каждое действие над данными должно быть авторизовано, проверено на соответствие прав доступа и задокументировано. Алгоритм проверки следующий:

Сообщение с запросом приходит на FES.

От FES делается запрос к серверу AS с целью узнать доступно ли конкретное действие с конкретными данными для ресурса/пользователя, от имени которого прислано сообщение.

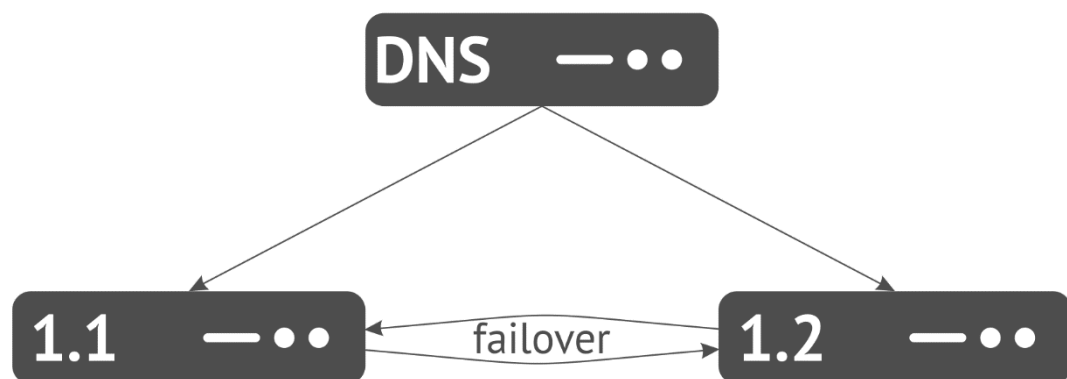
В случае если AS разрешает выполнять запрос или запросы, они передаются слою back-end или файловому серверу, в противном случае система возвращает ответ о невозможности выполнения запроса/запросов.

В случае попытки записать файл, сначала на вход FES приходит сообщение с запросом разрешения записи файла, в случае положительного ответа от AS, клиенту возвращается кодированная последовательность (на манер идентификатора сессии). На борту с этой последовательностью и с файлом для записи клиент затем стучится в файловое хранилище и если последовательность является легитимной, то файл будет сохранен.

Теперь рассмотрим каждую часть системы подробнее...

Передовая. Front-end

Уровень front-end содержит 2 сервера, которые превращают пришедшее на вход сообщение в запрос или запросы, а так же держат очередь этих преобразованных сообщений. Строго говоря, можно и не использовать 2 сервера сразу, а держать второй сервер только для failover'a в случае падения или компрометации основной ноды. Но поскольку сервера 2, то почему бы не использовать сразу и второй раз он все равно включен... Думаю имеет смысл сделать их полностью одинаковыми, получающими запросы по алгоритму Round-robin DNS, то есть поочередно: на уровне DNS каждый следующий запрос прокидывается то на первый, то на второй сервер.



Описание функционала:

1. Превращение сообщений в запросы

Изначально, сообщения были задуманы как способ скрыть конкретику запроса данных, а так же сократить тело запроса. Фактически сообщение это бинарная последовательность, которая может быть использована 2 способами:

Либо мы кодируем в сообщение текст и параметры запроса некой обратимой хэш-функцией;

Либо отправляем лишь идентификатор запроса и его параметры, так же в кодированном виде, а по прибытии на FES ищем соответствие идентификатора запроса с конкретным запросом в таблице возможных сообщений.

Первый способ можно использовать при условии что тексты запросов будут недлинными, либо хэширующая функция будет достаточно минифицировать тексты.

Преимущество первого способа может проявиться при наличии очень большого числа возможных сообщений, кое неминуемо образуется после более-менее серьезного срока эксплуатации системы.

Однако, даже при количестве запросов в миллион их беспрепятственно можно хранить в любом персистентном key-value хранилище, на ум, в первую очередь, приходит Redis,

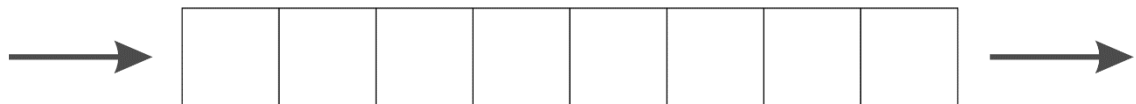
что даст нам скорость доступа практически не зависящую от количества элементов в хранилище. Таким образом второй способ выглядит предпочтительнее.

Тем не менее, при неблагоприятных условиях хранения, скажем, если все данные о миллионе запросов, по каким-то причинам, не будут влезать в память, использование этого способа может стать неоправданным.

В целом, я склоняюсь к использованию второго метода, так как его проблемы выглядят проще решаемыми, относительно проблем первого метода.

2. Очередь сообщений

Речь о том, что AS может не успевать проксировать или отклонять все поступающие к нему запросы. Логично продублировать AS и тем самым решить эту проблему. Собственно, от этого варианта никто и не отказывается, но априори хочется не обременять заказчика дополнительными тратами на второй сервер авторизации, а обойтись тем что есть. А на случай возникновения задержек при проксировании, сообщения будут складываться в очередь, предположительно организованную при помощи Redis.



Думаю имеет смысл создать пул постоянных соединений с AS и без усталости совать в них новые сообщения из очереди.

3. Heartbeat

Механизм HeartBeat позволяет анализировать работоспособность узлов в системе и автоматически изменять конфигурацию кластера, либо производить определенные действия в ответ на потерю одной или нескольких нод. В данном случае в механизме участвуют сервера 1.1 и 1.2.

Для этого можно использовать одноименный пакет.

Heartbeat обеспечивает основные функции, требующиеся для любой HA-системы, например, запуск и останов ресурсов, мониторинг доступности системы в кластере и передача прав владения общим IP-адресом между узлами кластера. Он следит за состоянием конкретной службы (или служб) по последовательному кабелю, интерфейсу Ethernet, либо по обоим. Поддерживает двухузловую конфигурацию, в которой для проверки состояния и доступности службы используется специальный квитирующий монитор heartbeat. Heartbeat предоставляет фундамент для более сложных сценариев.

Вроде через него можно даже load balancing делать, но эту функциональность использовать не будем.

Нам нужно следующее:

- 1) каждый из узлов уровня front-end должен как можно быстрее узнать что его «товарищ» в беде и не может отвечать на запросы;
- 2) как только узлу это становится ясно он должен попробовать реанимировать второй сервер (перезапустить сервисы, например), а если это не удалось – сообщить админу (про это подробнее ниже).

О проверке прав...

Заказчик хочет сделать сервер авторизации отдельным, и по приходу каждого сообщения запрашивать права на исполнения соответствующих команд. Гораздо быстрее работало бы решение с размещением небольшого хранилища данных о правах пользователей на выполнение запросов в front-end серверах без запросов к удаленному серверу. С другой стороны, есть мнение что такое решение небезопасно. Надо разобраться в этом вопросе.

Сделаем допущение что сервер авторизации на 100% недоступен извне кластера (в случае выделенного AS). Тогда единственный способ добраться до него (исключая физической соединение с сервером) это получить рут-доступ к FES и уже с него стучаться на сервер авторизации. Предотвратить исполнение каких бы то ни было команд, посланных злоумышленником с FES мы можем следующими способами:

- обрубить связь между скомпрометированным сервером и AS при возникновении подозрений у системы обнаружения вторжений (COB). Вот оказывается как, нужна еще и система обнаружения вторжений. Какая-то COB обязательно будет установлена, но ее отладка и настройка до оптимального реагирования процедура не мгновенная, да и гарантировать впоследствии ее безошибочное срабатывание нельзя;
- поключаясь к хранилищу мы проходим авторизацию под определенным пользователем. Так давайте дадим этому условному пользователю только те права, которые ему жизненно необходимы. В таком случае условный хакер пробравшийся на front-end машину не сможет разрушить сервер авторизации, а так же не сможет получить большую часть данных и инициировать их стирание и т. п. В таком случае, что изменения иных, защищенных параметром сервера и хранилища придется подключаться к серверу напрямую, что может быть неудобно, но зато безопасно.

В случае если хранилище данных авторизации лежит локально, то безусловно это решение может быть быстрее и надежнее при определенных условиях. Однако, условия эти вовсе необязательно должны иметь место. В случае хранения авторизационных данных на FES подключаться к хранилищу можно посредством Unix-сокета, при связи же VDS – VDS (виртуальные сервера) подключение может быть только с использованием сетевого (TCP) сокета что чуть медленнее и менее надежно + оверхэд на взаимодействие между виртуальными машинами. Если же сервера размещены на разных машинах, то встает вопрос как они соединены – если по Fibre

Channel или другому быстрому интерфейсу и машины находятся недалеко друг от друга (одна стойка, одна серверная, один ДЦ) то больших потерь от этого ждать не стоит.

Однако, как не крути, когда все внутри одной ОС взаимодействие будет быстрее, хоть эту разницу и можно свести к минимуму...

А вот потери в безопасности в этом случае могут быть совсем не маленькими, поскольку появление злоумышленника на машине с auth-данными, с рут-правами соответствующего хранилища или, что еще хуже, появления его в рут-правами системными может потенциально привести к компрометации и уничтожению всей системы.

Поэтому мы остановимся, все же, на конфигурации с отдельным сервером авторизации, как бы я тяготел к производительным системам, безопасность, в данном случае, важнее.

Давайте остановимся на безопасности и производительности front-end серверов и всей системы в целом, так как оные должны являться единственными точками входа в кластер из окружающего мира.

И так, у FES'ов открыты порт для подключений от серверов проектов (его номер не принципиален, но предположим что это 80-й) и порт для подключения по SSH (по умолчанию он 22-й, но лучше сменим на другой, например, на 22022).

80-й порт открыт только для определенных IP, на которых висят проекты использующие наш кластер, таким образом чтобы получить доступ к FES нужно залезть на один из «разрешенных» серверов или прикинуться одним из таких серверов, а затем организовать от имени этого сервера атаку по 80-му. Допустим на нем висит nginx и у нас есть оболоченный xpl0it, который умеет использовать некую уязвимость, переполнять буфер и выполнять произвольный код под рутом. Ситуация ужасная, но давайте попробуем сократить ее ужасность.

Выжечь все на подконтрольной машине можно, но кроме прекращения работы кластера это ничего не даст. Чтобы получить доступ к данным проектов нужно попасть на сервера уровня back-end или правильно к ним обратиться, чтобы накуралесить на сервере авторизации на него, так же, нужно еще попасть. AS, как уже было выше сказано, открыт только на подключения с FES под определенным пользователем, который ничего почти не может. Так что даже если удалось найти данные для подключения само это подключение ничего собственно не даст.

У меня есть идея сделать сервер авторизации шлюзом между front-end и back-end серверами. То есть, раскодированное сообщение попадая на AS проверяется на разрешенность и в случае положительного вердикта пробрасывается на уровень back-end, иначе возвращает FES сообщение о невозможности выполнения.

Таким образом чтобы получить какие-либо секретные данные, либо устроить их потерю с FES злоумышленнику надо долго и упорно бомбардировать AS комбинациями команд с параметрами дабы угадать комбинацию и быть пропущенным системой и все это во враждебной инфраструктуре с COB, рискуя быть «спаленным».

Все в основном крутится вокруг AS и его как можно более широкого канала на входе.

Канал действительно нужен широкий и стабильный дабы переварить трафик с уровня front-end без больших сетевых задержек.

Еще о сервере авторизации

Про этот компонент системы уже довольно много было написано. Если у нас будет 2 сервера AS, они по реализации во многом будут похожи на front-end сервера: 2 дублирующих сервера с failover'ом друг на друга и round-robin балансировкой, при падении одного вся нагрузка ложится на второй.

Хранилище авторизационных данных должно быть как можно более быстрым и поддерживать репликацию master-master. Структурирование данных в данном случае особо ни к чему, посему я в данном случае смотрю в сторону NoSQL-решений. Запрос авторизационных данных в хранилище скорее всего будет подразумевать попытку найти запись в БД, удовлетворяющую определенным критериям (запрашивающий пользователь, проект, устройство и т. п.), так что напрашиваются колоночные СУБД, но критериев все-такие не предполагается много, так что такое решение тоже выглядит избыточным. Думаю, использование key-value СУБД будет вполне оправданным, учитывая скорость выборки данных из таких систем. Redis как решение выглядит предпочтительным учитывая его распространенность и простоту, но Redis не поддерживает синхронную репликацию, по крайней мере пока. В качестве альтернативы есть, например, Riak, у которого вышеоговоренная репликация имеется, но Riak система не в пример сложнее и функциональнее Redis, и немного медленнее, а нам усложнение как бы ни к чему и скорость в приоритете. А master-master можно реализовать и на уровне bash-скриптов, например, или на Lua, как как аддон для Redis. На нем и остановимся пока.

На машине так же должна быть установлена COB следящая за неизменяемостью важных компонентов сервера.

Функциональность.

1. Авторизация запросов данных

Обращаемся к вышеописанному хранилищу при помощи, например, bash-скрипта или простого приложения и получаем простой ответ: «да» или «нет» (1 или 0, true или false, неважно). Далее пробрасываем транзакцию на back-end. Просто же...

2. Балансировка нагрузки серверов уровня back-end

Вариант 1. pgPool-II.

pgPool умеет балансировать select'ы способом Round-robin, таким образом мы всю запись производим на мастер, а выборки раскидываем между мастером и слэйвом.

Существует несколько типов балансировки: стандартная Round-robin балансировка на чтение в случае если сервера примерно одинаковы по производительности, либо Weighted Round Robin, если разнятся по скорости обработки запросов, либо балансировку Least Connections, что выглядит предпочтительным способом балансировки относительно предыдущих, особенно в случае неоднородной нагрузки. Однако, самым идеальным вариантом можно назвать балансировку со сбором статистики о каждом back-end сервере. Имеется ввиду, что балансировщик мониторит нагрузку, используемые ресурсы, время отклика каждого сервера и т. п., и исходя из этой информации принимает решение какому из серверов отдать на исполнение следующий запрос на выборку.

Подробнее ниже...

Вариант 2. HAProxy.



Позволю себе выдержку из Википедии...

«**HAProxy** – серверное программное обеспечение для обеспечения [высокой доступности](#) и [балансировки нагрузки](#) для [TCP](#) и [HTTP](#)-приложений, посредством распределения входящих запросов на несколько обслуживающих серверов...»

Утверждается что:

1U сервера оснащённые Xeon E5 (2014 года) и 10 Гбит/с сетевой картой без проблем обрабатывают поток 40–60 Гбит/с, при этом подчёркивается, что ограничивающим фактором является пропускная способность сетевой карты.^[10]

Даже на процессоре Intel Atom 1,6 ГГц (без воздушного охлаждения) HAProxy удалось обрабатывать поток до 1 Гбит/с.^[10]

Расход памяти: 1 Гб [ОЗУ](#) хватает для обслуживания ~20 000–30 000 одновременных сессий.

HAProxy умеет балансировать нагрузку на PostgreSQL и Redis (думаю, при использовании других СУБД он и с ними справится), переключать мастер при падении и снимать статистику о состоянии того или иного back-end сервера. Инструмент отличный

по функционалу и использованию ресурсов. Единственное что смущает это его редкое использование совместно с PostgreSQL – могут вылезти грабли.

Вариант 3. nginx.



Nginx без проблем проксирует запросы к PostgreSQL, Memcached, Redis и другим СУБД при подключении соответствующих модулей. Кроме того, nginx умеет распределять нагрузку, однако собрать статистику с проксируемых серверов и определить их нагруженность, в полном смысле этих выражений, без дополнительных скриптов не удастся, зато можно использовать следующие варианты балансировки:

round-robin (по умолчанию) – в объяснениях, я думаю, не нуждается – стандартный Round-robin алгоритм с поочередным задействованием всех отвечающих серверов в пуле;

weighted round-robin – Round-robin с весами – на выбор очередности выбора сервера влияет вес узла: узлы с большим весом обрабатывают запросы чаще;

hash - метод балансировки нагрузки для группы нод, при котором соответствие клиента серверу определяется при помощи хэшированного значения *ключа*. В качестве *ключа* может использоваться текст, переменные и их комбинации;

ip_hash - метод балансировки нагрузки, при котором запросы распределяются по серверам на основе IP-адресов клиентов. В качестве ключа для хэширования используются первые три октета IPv4-адреса клиента или IPv6-адрес клиента целиком. Метод гарантирует, что запросы одного и того же клиента будут всегда передаваться на один и тот же сервер;

least_conn - метод балансировки нагрузки, при котором запрос передаётся серверу с наименьшим числом активных соединений, с учётом весов серверов. Если подходит сразу несколько серверов, они выбираются циклически (в режиме round-robin) с учётом их весов;

least_time - метод балансировки нагрузки, при котором запрос передаётся серверу с наименьшим средним временем ответа и числом активных соединений с учётом весов серверов. Если подходит сразу несколько серверов, то они выбираются циклически (в режиме round-robin) с учётом их весов.

Методы `least_time` и `least_conn` с натяжкой можно назвать методами с мониторингом балансируемых серверов, хоть и очень простым.

Кроме того nginx может переключить мастер при его крахе (failover).

На этом можно, пожалуй, можно закончить рассмотрение этого компонента кластера, ибо я и так многовато рассказал :-)

Давайте теперь поговорим о файловом хранилище.

У меня было стойкое нежелание прогонять файлы через front-end сервер и через сервер авторизации, так что я придумал принимать решение о загрузке файла в хранилище до отправки самого файла. К тому же не плохо было бы не загружать файлы и на сервера проектов. Таким образом нам надо сделать авторизованную загрузку из браузера в кластер на сервер/сервера файлов (FS).

Предположим мы хотим загрузить новую видеолекцию на наш образовательный ресурс. Из веб-интерфейса проекта мы делаем запрос к серверу проекта, мол нам бы записать файл «Лекция по квантовой механике», в формате mp4, размером 240 Мб, с датой последнего изменения 24 января 2016 г. в 20:21 и хэш-суммой F6DE2FEA. Затем сервер проекта добавляет к запросу идентификатор пользователя, которым инициирована загрузка и пр. мета-инфу и отправляет запрос кластеру хранения. Запрос попадает на AS и в случае «законности» операции отвечает «Загрузку разрешаю», делает он это отправляя уникальный идентификатор загрузки, а параллельно скидывает сообщение файловому серверу «В течение 10 секунд к тебе начнет поступать файл «Лекция по квантовой механике», в формате mp4, размером 240 Мб, с датой последнего изменения 24 января 2016 г. в 20:21 и хэш-суммой F6DE2FEA». Сервер файлов сохраняет информацию о новом файле в очереди файлов и пишет информацию о файле в хранилище мета-информации. Сервер проекта получив положительный ответ сообщает его в браузер, где скрипт должен в ответ на это начать загрузку файла. Большие файлы должны быть поделены на части и могут отправляться следующими способами:

- последовательная отправка частей файла;
- параллельно, в любое доступное количество потоков.

Небольшие же файлы отправляются сразу одной частью. Решение о том, что считать большим файлом, а что маленьким принимает клиент – вместе с мета-информацией о файле он отправляет пометку что хотел бы отправлять файл по частям или что хотел бы отправить все одним куском. Преимущество «частичной» загрузки еще и в том, что при ее использовании можно организовать докачку файлов и индикацию прогресса загрузки: пока файл не полностью закачен или с последнего сеанса загрузки не прошло слишком много времени FS ожидает докачку всех частей файла, а прогресс изменяется с каждой новой закаченной частью.

Чтобы файловый сервер знал к какому конкретно файлу имеют отношение приходящие параллельно куски каждый кусок несет с собой идентификатор загрузки, описанный выше, как бы говоря FS: «Я кусок номер 3 от файла 176». Загрузка заканчивается, когда достигнут заявленный размер файла, после этого он собирается из

кусков в единое целое. Если итоговый размер файла больше заявленного – лишнее обрезается, а приславшему отправляется соответствующее сообщение.

Если какой-либо кусок приходит второй раз он замещает предыдущий.

После полного получения файла, запись о нем удаляется из очереди файлов, а все необходимые данные пишутся уже в персистентное хранилище.

Каждый записанный файл проверяется на соответствие заявленной хэш-сумме: демон в фоне рассчитывает хэш каждого нового файла и сравнивает ее с сохраненной суммой, заявленной для этого файла (той что приходила в запросе на загрузку). Если хэши не совпали файл удаляется, а в базу данных о файлах пишется соответствующая пометка.

Стоит отметить что природа операции по записи в FS транзакционна. То есть, при неудачной записи файла (по любым причинам) данные об этом файле с БД будут откатаны, а вот при неудачной записи мета-данных в базу сервер опробует еще пару раз это сделать и только потом запретит загрузку файла.

Если касаться аппаратной части, то желательно использование твердотельных накопителей в аппаратном RAID 10 с батареей, ибо сеть, согласно статистике, уже обогнала обычные HDD, а писать нам надо как раз из сети и желательно без задержек. Но на самом деле это не так принципиально – если вместо SSD будут серверные HDD нас выручит RAID даже при немалых нагрузках. Судите сами, какой должен быть входящий поток, чтобы повесить систему, если у нас есть 8 дисков в RAID 10 с кэшем на 1Гб :-). Получается мы пишем и читаем одновременно в 4 потока: по 1 потоку на пару дисков-зеркал, сразу на оба диска. Если приравнять скорость сети к скорости серверного HDD (и это еще очень грубое допущение, так как это передача по внутренней сети, а то и через WWW, на самом деле скорость загрузки будет много меньше скорости диска, по крайней мере в России), то получается мы можем писать и читать параллельно 4 файла совсем без задержек. А ведь есть еще гигабайтный кэш. На вскидку не скажу на каком количестве параллельно записывающихся файлов будет ощущаться «заторможенность», но думаю это число не меньше 50, особенно на суд нашего, неизболованного скоростью загрузки, российского пользователя. Большие и редкозапрашиваемые файлы можно не пускать в кэш дабы не забивать его

Нужно ли такой машине много оперативной памяти? Скорее нет, чем да. Положим, гигабайт 8 можно выделить под частоиспользуемые небольшие файлы и на этом хорош.

Для еще большей производительности на запись желательно писать файлы последовательно, тогда не будет тратиться время на перемещение головки диска. В этом случае нам бы помог Naustack, но как по мне, для использования таких решений они должны быть действительно необходимы.



Haystack - проект Массачусетского технологического институт, используется в Facebook для хранения тонны картинок, генерируемых огромной армией пользователей социальной сети. Не вдаваясь в подробности стоит сказать что основное ноу-хау этой системы – возможность последовательно писать и читать с HDD.

Очень крутая вещь, но открытой реализации нет с общим доступе. Есть только идеи которые можно реализовать своими силами, что пока видится нерентабельным, так как пока не понятно будет ли подсистема хранения файлов работать под такой нагрузкой, чтобы решения типа Haystack давали реальный прирост в производительности.

Репликация FS на резервную копию в Google Drive, в идеальном случае, должна происходить во времена относительной незагруженности канала FS. Более-менее это время можно спрогнозировать по статистике, но всегда есть вероятность что синхронизация случится, все же, далеко не в лучшее время и повлияет на загрузку новых файлов в FS и чтение. Думаю, лучшим способом будет выделить, к примеру, 20% канала на синхронизацию и размазать нагрузку на все время. Таким образом основная задача по закачке новых файлов от клиентов и чтения уже закачанных хоть и потеряет 20% пропускной способности, но зато этот канал будет всегда, как и канал синхронизации. Приведенное мной соотношение 80/20 можно варьировать в зависимости от ситуации с производительностью – не будет ли одна из задач излишне запаздывать относительно другой. Изменение соотношения можно изменять и автоматически, если снимать соответствующие сетевые показатели, скажем, раз в 5 минут и в зависимости от них менять баланс.

Кроме того, некоторые файлы, потеря которых не критична, можно вовсе не бэкапить. Тогда канал на выгрузку в облачное хранилище будет экономиться.

Как обстоят дела с чтением файлов?

Почти так же как и с чтением текстовых данных: шлём сообщение с клиента о том что хотим получить файл, если нам это разрешено, то получаем редирект на местонахождение этого файла на FS.

А откуда этот редирект?

Сервер авторизации при соответствии прав на файл спрашивает ссылку на него у файлового сервера.

Если же файл потерялся, был удален и т. п., то FS возвратит ошибку, которую AS бережно передаст клиенту. Может случиться так что на FS файл был утерян (не удален, это большая разница), а на бэкапе он в наличии. Сгенерировать ссылку на бэкап не получится, так как Google Drive не даст получить закрытые от внешнего мира файлы без авторизации, а авторизованным для резервного хранилища у нас является только основной FS. Все файлы в гугль-хранилище, сохраненные от имени FS являются именно закрытыми, думаю, понятно почему. В таком случае мы должны ответить что файл временно недоступен и немедленно запустить восстановление его из бэкапа. За самим файлом мы можем сходить в Google Drive, а вот чтобы восстановить данные БД нам так же нужен и бэкап БД. Где разместить этот бэкап? Отдельный сервер городить под эту задачу не хочется. Для начала будет оставлять бэкапы все на том же FSB и дополнительно складывать в FSB.

Что делать если файл побился?

В случае такой нехорошей ситуации, мы ее должны, в первую очередь, идентифицировать: предполагается запуск дополнительного демона, который в фоне будет пересчитывать хэш-суммы файлов и сравнивать их с сохраненными в базе данных. По очереди, отсортированной по сроку без проверки и обновлений, то есть сначала те, которые давно не проверялись и не обновлялись. Возможно использование и других алгоритмов, например, основанных на размере файла (чем меньше файл, тем быстрее считается его хэш, так может сначала проверять маленькие файлы?), частоте запрашивания и изменения файла, а так же алгоритмических гибридов. Важно понимать что такой демон не решит всех проблем, он только снизит вероятность нахождения на сервере битого файла, кроме того он будет отъедать ресурсы и его бесконтрольное использование может сократить производительность основных компонентов. Так что тут нужны тесты, на основе которых нужно найти количество выделяемых ресурсов, при котором работа демона не будет бить по работоспособности других систем.

Удаление файлов. Эта операция должна быть атомарна в пределах файлового хранилища:

1. Удаляем файл.
2. Удаляем запись в БД или помечаем неактивной.
3. Отправляем сигнал FSB удалить файл из бэкапа.

Операции 1.1 и 1.2 можно выполнять параллельно. При возникновении ошибки в любой из операций все уже совершенные откатываются.

Но как это сделать? В смысле как откатить удаление уже удаленного файла? В Linux, например, есть такое решение <https://github.com/sindresorhus/trash-cli> - реализация корзины с управлением из командной строки. Так же всем известны пляски с алиасом на `rm` – не красивое, но рабочее решение.

Самое интересное. Back-end сервера.

Запросы к этому уровню попадают от AS, в случае если он разрешает получение или изменение данных.

На каждом сервере необходимо поднять пул соединений к базам данных. Строго говоря, его использование, не смотря на, вроде бы, очевидную пользу не всегда позитивно влияет на производительность доступа к данным в плане повторного использования соединений. Вроде бы, это же хорошо когда не нужно на каждую транзакцию создавать новое соединение с БД и процесс (если это PostgreSQL): но все же дополнительный слой взаимодействия в виде пула может вносить overhead, а его преимущества могут практически повторяться использованием постоянных соединений клиентов напрямую к БД (не актуально для PHP). В нашем случае, у нас только один узел подключается к базам данных – AS, если он будет иметь rconnect'ы со всеми базами, возможно, это будет быстрее, чем использование пула соединений, в будущем нам ждут тесты на этот счет, но по тестам с увеличением нагрузки прирост производительности от задействования пула становится весьма ощутимым.

Оптимальным вариантом для реализации выглядит pgBouncer. Он не требователен к ресурсам и не несет с собой практически никакого overhead, но в случае активного использования pgPool-II для других целей имеет смысл отказаться от pgBouncer как раз в пользу данного функционала pgPool-II, дабы не создавать лишний слой абстракции, а с ним и точку отказа и возможного увеличения latency.

Оба решения имеют функционал для организации очереди запросов: если в какой-то момент все соединения из пула заняты, пришедший на вход запрос может быть добавлен в очередь и будет ждать свободного соединения.

И так, запрос (транзакция) попадает на условный порт X на котором живет pgBouncer, в случае наличия свободных соединений к БД в пуле мы забираем одно из них и выполняем запрос, иначе ждем освобождения соединения в очереди. По большому счету это все что делает сервер. Его задачи – делать свою работу быстро и бесшбойно. Машине (желательно физической), при возрастающей нагрузке, будет необходимо все больше памяти для соответствия строгим требованиям по производительности. Диски логично выбрать типа HDD, так как они пока еще остаются более надежными нежели твердотельные. А вот на скорости вращения лучше не экономить 15 000 об./мин. самое то. Так же нам понадобится аппаратный RAID 10 с батареей и достаточным кэшем, при его наличии можно пренебречь кэшем дисков.

Пара BES будет объединена как master-slave, с потоковой репликацией.

Чаще всего данные реплицируются асинхронно, но для данных, актуальность которых важна будет приписана синхронная потоковая репликация.

При падении мастера должен происходить failover на slave. Потери некритичных данных при этом вполне возможны.

Если подробнее. То потери возможны на данных, которые не успели отреплицироваться на slave. При падении мастера они будут утеряны. В время неработоспособности мастера все запросы будут идти на новоиспеченный мастер, бывший слейв. Соответствующие изменения должны произойти в конфигурации балансировщика нагрузки, точнее балансировку следует на время прекратить и слать все запросы по одному адресу. После подъема «настоящего» мастера он станет слейвом.

Отдельно стоит рассказать о сервере 2.3.

Некоторые особо важные изменения данных, например, финансовой информации, учетных данных пользователей и т. п. будут дублироваться на сервере 2.3. Как вариант, можно использовать логическую репликацию определенных таблиц, хранящих важные данные. Проще говоря, после записи данных в мастер (например, сервер 2.1) они должны быть записаны на сервер 2.3 и пока они туда железно не записались мастер не может ответить что данные записаны. Этот сервер должен быть прежде всего надежным. Значит мне хотелось бы видеть на нем аппаратный RAID, например, 61 с батареей. Хранилища, будь это PostgreSQL и что-то другое, должны быть настроены на максимальную надежность, это значит запись, по возможности, сразу на диск – упреждающая запись.

Безопасность. Сервер 2.3 работает только с одним открым портом для репликации от 2.1 (мастера). 2.2 ждет подключений с 2.1 на репликацию и с AS для выполнения запросов на чтение. 2.2, в свою очередь, может принимать запросы от AS и на чтение и на изменение.

Поговорим о серверах хранения логов.

Я считаю что логи со всех основных машин лучше хранить в одном месте что даст возможность ими удобно управлять и анализировать.

Систем сборки логов сейчас великое множество. Например, Scribe или Fluentd.

Работа большей части из них сводится к демону, работающему на сервере приложения и следящему за изменениями в файлах логов приложений. Как только изменения появляются они отправляются на машину-хранилище логов. Многие системы еще и анализируют логи и предоставляют обработанную информацию в графическом виде, что может быть очень удобным.

Никаких «велосипедов» здесь не планируется. Берем готовое, желательно, бесплатное решение и используем его. Будет здорово, если будет возможность допилить кое-какие аспекты под себя.

Трафик на машину с логами будет бешеный как и нагрузка на диски. Нам нужна будет машина с быстрыми дисками в RAID'e + ее связь со всеми машинами кластера, отсюда еще один дополнительным открытый порт на всех нодах. Поскольку частичная потеря

логов не критична, то можно воспользоваться треддотельными накопителями. К тому же у нас будет еще и сервер бэкапа логов. На нем мы можем использовать, например, RAID 60 из старых-добрых HDD, если даже запись будет запаздывать - не беда. А вот на основном сервере логов (LS) лучше юзать RAID 50 для удобоваримой скорости и неплохой надежности. Реплицировать логи с LS на LSB я собираюсь при помощи обычного rsync.

Бесполезная нагрузка. Бэкап уровень.

Была бы моя воля, я бы, конечно, оставил этот уровень. Вот только в нем был бы всего 1 сервер. Но имеем то что имеем.

По задумке, на каждом сервере концентрируются данных исходя из их отношения к определенному проекту или организации (как точно пока не определились) + некоторые данные, как то данные пользователей, стоят особняком, так как к ним обращаются все проекты, для них тоже нужен отдельный RS.

Кстати, если у кого-то возникнет вопрос почему машины уровня backup называются RS? Ведь по логике предыдущих сокращений они должны зваться BS. Но нет. В терминах заказчика эти сервера зовутся «сервера ресурсов» (resource servers), так что RS.

Так вот. На машинах уровня back-end все данных, хранящиеся в PostgreSQL (а таких большинство) лежат в одной базе, для более простых и быстрых join'ов без FDW, логически поделенной на схемы по такому же принципу как поделены данные между серверами ресурсов. Таким образом, нам нужно реплицировать (именно реплицировать, а не бэкапить, ибо данные на RS должны быть живыми – лежащими в базе данных, готовыми к использованию) каждую схему на отдельный RS. Средствами стандартной и всеми любимой потоковой репликации тут никак не справиться, так как она не дает возможности реплицироваться отдельные схемы базы. Нужна логическая репликация.

Можно сказать хорошей логической репликации в PostgreSQL нет. В 9.5 появилась логическая репликация из коробки - Logical Decoding, но назвать ее полноценной язык не поворачивается, пока это сырое решение, хоть и с перспективой.

Вот, например:

Logical Decoding требует записи дополнительной информации в WAL. Не критично, но неприятно;

никакой параллельности, каскадная репликация не поддерживается.

Покрутим-попробуем, может подойдет. Тем более 9.5 уже вышла, так сказать, больше стабильности по едее...

Есть так же сторонние уже потертые решения а-ля Slony, Bucardo, Londiste, pgPool-II, которые по слухам не блещат производительностью или многообещающая разработка

BDR, которую собираются запилить в postgres в одной из следующих версий. В случае некошерности Logical Decoding будем пробовать прикрутить pgPool-II, процесс этот обещает быть не сложным и в будущем не приносить хлопот: на pgPool много документации, статей, примеров его производительность и стабильность не хуже аналогов, к тому же мы можем для пула соединений, очереди и репликации на RSы использовать только одно приложение – pgPool-II, а pgBouncer выкинуть. Скорость репликации нам в данном случае не принципиальна, хотя, безусловно, чем быстрее тем лучше. Но, само собой, скоростей потоковой репликации нам не достичь. Ну и ладно. Вряд ли «отставание» данных на RSax будет для кого-то критичным.

Над аппаратной начинкой серверов ресурсов нам, видимо, корпеть не придется, все на совести «хранителей» конкретного RS, пусть исходят из своих задач, а мы если что подскажем.

Чуть не забыл про сервер BS. В нужности которого я сомневаюсь. По идее там будет храниться SQL-бэкап базы с 2.1 (или 2.2). Вроде бы зачем он нужен если у нас и так 2 дублирующих сервера на back-end уровне (а для некоторых данных все 3). Но есть вероятность, что побитые данные с 2.1 перетекут потоком на другие сервера уровня и все станет не хорошо. Спокойнее иметь бэкап. Можно сохранять его на BS, например, раз в сутки и хранить бэкапы за последнюю неделю. Разумеется все заархивированное.

Отступление: про СУБД отличные от PostgreSQL.

Зоопарк СУБД пока не утвержден. Известно лишь что будут Postgres и Redis. Есть задача для которой хорошо подойдет колоночная система на манер Vertica или InfoBright. И неизвестно что еще придет в голову добавить в качестве хранилищ данных. Хорошо отношусь к Tarantool, но задач под это решение пока нет, что может и хорошо.

Все вышеупомянутые СУБД умеют растекаться на кластер, бэкапиться, быстро и надежно работать, имеют армию приверженцев и хороший багаж проектов с их участием. Так что не пропадем! :-)