



Издательство
Олега Бунина

Разработка высоконагруженных систем

По материалам конференции  HighLoad++
2010-2011

Сборник лучших докладов конференции
разработчиков высоконагруженных систем

HighLoad++ за 2010 и 2011 года

**Если Вы хотите получить консультацию этих или любых других профессионалов, специально подобранных под Вашу задачу;
если перед Вами стоит задача разработать сложный
или высоконагруженный проект — обращайтесь:**

**Олег Бунин
oleg.bunin@ontico.ru
+7 (916) 635-9584**

Мы поможем :)

Оглавление

Академия высоких нагрузок

Учебник по высоким нагрузкам. Урок первый	5
Урок второй. Масштабирование фронтендов	15
Урок 3. Масштабирование бекенда	24
Урок 4. Масштабирование во времени	31
Урок 5. Базы данных. Последний пункт обязательной программы.....	38

Секция «Архитектуры»

Впервые в рунете: сказ о 100М писем в день / Андрей Сас.....	49
Высокая нагрузка на erlang-приложения: erlyvideo на гигабитном канале / Максим Лапшин.....	73
Специализированные http-демона: круг решаемых задач, подходы и метод / Сергей Боченков, Александр Панков	89
Как мы храним 75 млн пользователей (пишем неблокируемый сервер) / Денис Бирюков	100
Как построить высокопроизводительный front-end сервер / Александр Крижановский	117
Pconnect: граната в руках обезьяны / Сергей Аверин.....	133
Низкоуровневая оптимизация C/C++ / Андрей Аксенов.....	150
Анатомия баннерной системы / Артем Вольф труб.....	176
Node.JS / Андрей Костенко.....	194
AJAX Layout / Олег Илларионов	207
Интеграция открытых технологий и взаимодействие со сторонними проектами в условиях высоких нагрузок / Олег Илларионов	233
Круглый стол ВКонтакте / Олег Илларионов, Павел Дуров	250
Архитектура новой почтовой системы Рамблера / Андрей Шетухин	264

Секция «Системы хранения»

Почему не стоит использовать MongoDB / Сергей Туленцев	275
12 вариантов использования Redis — в Tarantool / Александр Календарев, Константин Осипов	282
Управляемый code injection: как мы считаем все пользовательские отчеты за один проход в системе интернет-статистики Openstat / Михаил Якшин	301
Архитектуры Backup & Recovery решений / Илья Космодемьянский	319
Архитектура хранилища бинарных данных на Одноклассниках / Александр Христофоров, Олег Анастасьев.....	338
Нестандартное использование репликации Mysql / Дмитрий Самиров, Александр Панков....	358

Секция «Системное администрирование»

Принципы балансировки / Алексей Бажин	367
Управление памятью в гипервизоре. Все о виртуализации памяти в Parallels / Анна Воробьева	380
Большая книга рецептов или часто задаваемые вопросы по управлению сложными системами / Александр Титов, Игорь Курочкин.....	403

Академия высоких нагрузок

Впервые опубликовано в журнале Хакер (www.xakep.ru)

Олег Бунин, Максим Лапшин, Константин Осипов и Константин Машуков

Каждый программист хочет стать лучшим, получать все более интересные и сложные задачи и решать их все более эффективными способами. В мире интернет-разработок к таким задачам можно отнести те, с которыми сталкиваются разработчики высоконагруженных систем.

Большая часть информации, опубликованная по теме высоких нагрузок в интернете, представляет собой всего лишь описания технических характеристик крупных систем. Мы же попробуем изложить принципы, по которым строятся архитектуры самых передовых и самых посещаемых интернет-проектов нашего времени.

Учебник по высоким нагрузкам Урок #1

От авторов

Основным направлением деятельности нашей компании является решение проблем, связанных с высокой нагрузкой, консультирование, проектирование масштабируемых архитектур, проведение нагрузочных тестирований и оптимизация сайтов. В число наших клиентов входят инвесторы из России и со всего мира, а также проекты «ВКонтакте», «Эльдорадо», «Имхонет», Photosight.ru и другие. Во время консультаций мы часто сталкиваемся с тем, что многие не знают самых основ — что такое масштабирование и каким оно бывает, какие инструменты и для чего используются. Эта публикация открывает серию статей «Учебник по высоким нагрузкам». В этих статьях мы постараемся последовательно рассказать обо всех инструментах, которые используются при построении архитектуры высоконагруженных систем.

Учебник по высоким нагрузкам. Урок первый

Мы начнем описывать построение архитектуры высоконагруженных систем с самых основ. Не будем рассказывать ни о каких ноу-хау и постараемся не разделять возможные решения на «правильные» и «неправильные». Конечно, у нас есть излюбленные концепции, однако мы планируем дать наиболее полное представление о целом наборе приемов, методов, подходов, схем и инструментов, которые можно использовать.

В рамках нашего цикла мы будем объяснять, зачем нужен тот или иной инструмент, что он умеет делать, в чем может помочь, какие подводные камни могут возникнуть при его использовании, куда дальше копать и т. д.

Монолитные приложения и сервис-ориентированная архитектура

Итак, прежде всего нужно определиться с терминологией, чтобы правильно понимать друг друга. Рассмотрим два принципиально разных подхода к построению архитектуры веб-проектов.

Монолитное приложение

Приложение представляет из себя монолитный программный код.

Плюсы:

- Отсутствие какого-либо оверхеда на интеркоммуникацию сервисов;

Минусы:

- Высокая сложность разработки, кадры решают все;
- В случае проблемы встает все;
- Невозможность вести распределенную разработку.

Монолитное приложение — это один большой кусок. Такие приложения могут работать очень быстро, поскольку в них нет никаких оверхедов, связанных с тем, что данные перегоняются из одного блока в другой, от одного сервиса к другому или каким-то образом конвертируются.

Один из самых известных примеров монолитного приложения — крупнейший почтовый сервис CommuniGate Pro. Во всяком случае, несколько лет назад он был монолитным и очень быстро работал. Однако он представлял собой немасштабируемое приложение, которое тем не менее вполне держало миллион пользователей.

Оно было написано одним человеком, и в какой-то момент это стало проблемой. Подключить новых программистов сравнимого уровня оказалось почти невозможным. Это основная беда монолитной архитектуры — большие сложности в масштабируемости разработки. Речь идет именно о распараллеливании разработки, а не о масштабировании программного решения.

Гораздо чаще в интернете используется так называемая сервис-ориентированная архитектура. Она подразумевает разделение программного обеспечения, сайта на некие сервисы, каждый из которых отвечает за что-то одно. При этом все сервисы обмениваются друг с другом данными по определенному протоколу.

Сервис-ориентированная архитектура (SOA)

Каждый сервис решает строго определенную задачу. Основной минус этого подхода заключается в наличии оверхеда на интеркоммуникацию сервисов между собой и на обработку API взаимодействия между слоями.

Рассмотрим, например, Facebook. Он построен почти классически. Есть различные сервисы, каждый из которых реализует строго определенный набор функций. К примеру, служба со-

общений и ленты новостей (то, что, казалось бы, является ядром сети) представляют собой отдельные сервисы, которые тесно интегрированы. Возьмем сервис авторизации. Мы можем обратиться к нему, передать ему куку и спросить: «Это валидная кука? Если валидная, то кому она принадлежит?»

Наверное, одним из самых известных проектов, при построении которого сервисный подход используется по максимуму, является Amazon. Этот большой интернет-магазин сталкивается с задачами, которые характерны для любого крупного проекта. Когда в компанию пришел новый технический директор, он принял гениальное решение: «Мы будем делать сервисы!» В результате Amazon является не только и не столько крупным интернет-магазином, сколько поставщиком cloud-сервисов. Теперь при создании какого-либо продукта, крупного сайта всегда возникает вопрос: разрабатывать ли собственную систему хранения, или использовать систему от Amazon, которая изначально построена так, чтобы ее можно было купить?

Один из самых больших плюсов сервис-ориентированной архитектуры — возможность вести распределенную разработку. Тут надо понимать, что под распределенной разработкой имеют в виду вовсе не такую разработку, когда члены команды находятся в разных точках земного шара — один в Таиланде, а другой в Москве. Ситуация намного шире. Скажем, вы наняли в Москве пятнадцать программистов, а шестнадцатого нанять не можете. Пока вы нанимаете шестнадцатого, первый уже увольняется. Или другой пример. Когда возникают какие-либо ограничения, связанные с командой, некоторые задачи приходится передавать аутсорсерам. И эта другая команда, состоящая из незнакомых людей, которой вы уже не можете управлять, начинает коммитить что-то прямо в ваш репозиторий, туда же, куда и ваши основные девелоперы. Это проблема, которую очень сложно решить. Именно поэтому «монолитный» подход к монолитному приложению осложняет масштабирование разработки, когда речь, допустим, идет о едином PHP-приложении. Например, обычный CGI script в каком-то смысле является монолитным приложением. Когда нужно, чтобы этот CGI script «пилило» несколько человек, уже начинаются трудности.

Если у вас сервис-ориентированная разработка, вы можете прийти к сторонней команде и сказать: «Ребята, нам надо запилить эту штуку. Мы API продумали, оно должно быть таким». И можно действительно кусок вашего приложения отдать на разработку другим людям. Легко может оказаться, что задача уже давно кем-то решена и доступна в виде готового для использования решения.

Проект именно так и эволюционирует: берется монолитное приложение, разбивается на отдельные сервисы, каждый из которых отвечает за строго определенный набор задач, после чего для этих сервисов задается способ коммуникации. Он может быть каким угодно: от REST API по HTTP до простых запросов к базе данных. Неважно, что именно используется в качестве общей шины.

Другой пример: софт Erlvideo, разработкой которого занимается Максим Лапшин. Это среднего размера софтина на языке программирования Erlang. Писать подобные видеостриминговые штуки сложнее, чем сайты, потому что надо одновременно держать в голове множество тонкостей. В такие проекты всегда трудно привлекать людей, а на обучение программиста, который бы смог написать то, что не сломало бы код на продакшне при выкатывании, нужно минимум полгода. В случае с «Эрливидео» та самая проблема двух гениев решена инструментально. Инструмент помогает вести параллельную разработку в одном коде, в одном приложении, в одном репозитории. Однако это, скорее, исключение из общего правила, обусловленное компактностью кода.

Ремесленный и промышленный подход

Условно говоря, существует два подхода к разработке высоконагруженных систем: ремесленный и промышленный. В чем разница? В том, где разрабатываются средства масштабирования — те инструменты, которые гарантируют, что ваша система будет выдерживать огромные нагрузки.

При использовании промышленного подхода эти средства разрабатываются отдельно от бизнес-логики. При ремесленном подходе средства и бизнес-логика разрабатываются одновременно. И там, и там имеются сервисы. Но в одном случае есть один большой слой, который отвечает за то, что все это будет работать. Объясним на примере.



Разработчики Google в большинстве своем не специализируются на разработке высоконагруженных систем. Если у кого-нибудь из них спросить: «Как ты будешь делать эту систему? Как она будет выдерживать миллионы пользователей?», то, скорее всего, услышишь: «Это очень просто. Я сделаю запрос к системе big data, и она быстро вернет ответ». Он не знает, что происходит внутри — ему это не надо. Если посмотреть на схему, то он работает на «зеленом» уровне и использует уже готовые разработки, сделанные отдельной командой инженеров, которые непосредственно занимаются высокими нагрузками. Так работает большинство крупных компаний.

Хорошим примером приложения, при разработке которого использовался этот подход, может послужить Google+. Это приложение было создано за совершенно фантастический срок в пару месяцев и приняло на себя, наверное, одну из самых чудовищных нагрузок по росту числа пользователей за всю историю интернета.



Аналогичным образом устроено большинство крупных проектов — Facebook, Google, «Яндекс». В «Яндексе» тоже есть эта волшебная шина данных, к которой идут запросы, и «Ян-

декс» никогда не падает целиком, за исключением тех случаев, когда возникают проблемы с data-центром. В «Яндексе» падают куски страниц. Почему? Потому что какой-то сервис или какое-то хранилище перестает отвечать. Если, например, упал сервис погоды, то на главной странице будет отображаться все, кроме погоды. Именно при таком подходе одни специалисты отвечают за масштабирование, сборку, коммуникацию, а другие программируют, к примеру, отображение погоды или курса валют.

В качестве примера сайта, созданного с использованием ремесленного подхода, можно привести «ВКонтакте». Когда для «ВКонтакте» разрабатывается какой-либо отдельный сервис, например чат, разработчику передают все полномочия, специально оговаривая, что этот чат должен быть масштабируемым.



Независимо от того, имеются ли общие примитивы по хранению данных и прочее, в целом разработчик самостоятельно принимает решения. Он работает не только над самой бизнес-логикой, но и над ее масштабированием.

Плюсы и минусы ремесленного подхода интуитивно понятны. Если вы хотите использовать такой подход, вам опять же нужны очень хорошие, гениальные программисты, вам нужны люди, которые досконально разбираются во всех тонкостях, в том, что и как работает. Если спросить Павла Дурова, сколько человек из его команды обладает навыками по созданию высоконагруженных систем, он ответит: «Все!»

Помимо высоких требований к квалификации, которые не позволяют, например, расшириться в два-три раза, для ремесленного подхода характерно также отсутствие оверхеда на общую шину, по которой гоняются общие данные, и максимально полное и эффективное использование машинных ресурсов. Когда «ВКонтакте» пару лет назад создавал чат по аналогии с Facebook, у него, по-моему, было всего две-три машины с установленным ejabberd-сервером.

Ремесленный подход

- Быстрая разработка любых новых решений;
- Высокие требования к квалификации разработчиков – низкая масштабируемость разработки;
- Максимально эффективное использование технологий и аппаратного обеспечения;



Точно так же дело обстоит и с поиском. У «ВКонтакте» отдельный сервис поиска — очень быстрый, его переписывали множество раз. Хотя он очень большой, им занимается всего несколько человек, которые выполняют все необходимые операции вручную. Нет никакого машинного сервиса, которому можно сказать: «Отдай индекс и разложи его на 100 серверов». Они делают это сами, руками.

Что касается повышенных требований к аппаратному обеспечению, то, если мы говорим о промышленном подходе, крупному бизнесу куда предпочтительнее вложить заранее известную сумму, пусть даже и очень большую, и получить за нее необходимый рабочий функционал. Некоторые считают, что это слишком дорого. Гораздо дешевле будет нанять дорогих специалистов, которые создадут систему, занимающую не тысячу серверов, а только 20. Те, кто придерживается этой позиции, утверждают, что тысяча серверов — это слишком высокая плата за безболезненное масштабирование, поэтому лучше нанять людей, которые будут следовать ремесленному подходу, используя уникальные инструменты.

Остановимся на еще одном немаловажном преимуществе ремесленного подхода. Дело в том, что крупные компании часто являются пионерами. Допустим, DynamoDB был разработан в Amazon для масштабирования их собственной системы работы со складом. После появления DynamoDB оленесорским сообществом были написаны Cassandra, Hadoop и так далее. Все они существуют благодаря компаниям, использующим ремесленный подход, которые нуждаются в собственных сервисах типа DynamoDB, но не в состоянии разработать их самостоятельно. Таким образом, подобный обмен происходит постоянно. Google создает какое-то новое решение. Это решение опенсорсится, потом много раз переписывается, после чего его подхватывают десятки компаний, таких как «ВКонтакте», DeNA (Япония) и многие другие. Они делают из этого решения действительно качественный софт, которым могут пользоваться все.

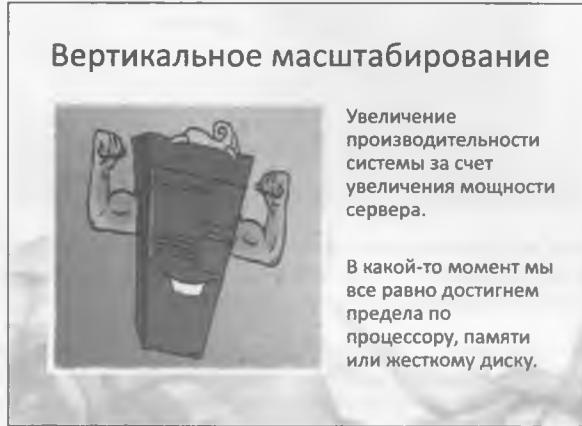
У крупных компаний из-за этого часто возникает предубеждение, которое можно выразить фразой «Все, что сделано не нами, нам не подходит» или «Not invented here». Крупные компании могут себе позволить привлекать профессоров и целые университеты, чтобы они вели какое-либо направление. Например, Google translate курирует знаменитый профессор, помоему, из Стэнфорда, который всю жизнь занимался теорией машинного перевода. То же самое происходит и в Microsoft, и других компаниях такой величины.

Профессионалы есть в компаниях обоих типов. Но это разного рода профессионалы. Профессионалы в тех компаниях, которые практикуют ремесленный подход, действительно знают, какие оленесорные средства нужно прикрутить, а что написать самим, чтобы все заработало прямо завтра. В больших корпорациях, скорее всего, есть не только гуру, которые сидят и «пилят» big data и web scale, но и огромное количество специалистов, которые занимаются инновациями именно в области usability, новых сервисов и прочего.

Далее мы будем говорить в основном о сервисной архитектуре и об инструментах, которые можно применять в масштабировании.

Масштабирование архитектурного решения

Для начала рассмотрим самые основы — вертикальное и горизонтальное масштабирование. В чем состоит концепция вертикального масштабирования?



Вертикальное масштабирование заключается в увеличении производительности системы за счет увеличения мощности сервера. По сути, при вертикальном масштабировании задача увеличения производительности отдается на аутсорс производителям железа. Специалисты, которые делают большую железку, предлагают некое обобщенное решение, которое будет работать быстрее. Вы заменили машину — и у вас стало работать быстрее.

Какие здесь есть опасности? Один из главных недостатков — вертикальное масштабирование ограничено определенным пределом. Параметры железа нельзя увеличивать бесконечно. В какой-то момент станет нужна уже тысяча серверов. Закупить столько железа будет либо невозможно, либо нецелесообразно. Кроме того, стоимость машин с более высокими характеристиками не обязательно возрастает линейно. Следующая по мощности машина может стоить уже даже не в два раза дороже, чем предыдущая. Поэтому вертикальное масштабирование требует крайне аккуратного планирования.

Альтернативный подход — горизонтальное масштабирование

Горизонтальное масштабирование



Увеличение производительности системы за счет подключения дополнительных серверов

Основной его принцип — мы подключаем дополнительные серверы, хранилища и учим нашу программную систему использовать их все. Именно горизонтальное масштабирование является сейчас фактически стандартом.

Однако на самом деле вертикальная компонента присутствует практически всегда, а универсального горизонтального масштабирования как такового не существует. Известен также такой термин, как диагональное масштабирование. Оно подразумевает одновременное использование двух подходов, то есть вы сразу и покупаете новое железо, тем самым выигрывая время, и активно переписываете приложения. Например, такой подход принят в Stack Overflow.

И еще один способ масштабирования. Как вообще выполняется любое масштабирование, как проектируются высоконагруженные системы? Первое, что необходимо сделать, — это изучить предметную область, как данные движутся внутри системы, как они обрабатываются, откуда и куда текут. Интернет-проект — это в каком-то роде система для различного представления разных данных с разными свойствами.

При этом некоторые данные всегда должны быть актуальными, а на другие можно «забить» и показывать их обновление не сразу. Приведем простейший пример. Пользователь, который постит сообщение в свой блог, должен тут же увидеть это сообщение, иначе он подумает, что что-то сломалось. А во френд-ленте друзей пользователя это сообщение может появиться и через минуту.

Зная такие особенности, можно применять прекрасный метод, который называют отложенной обработкой. Его суть заключается в том, что данные обрабатываются в тот момент, когда это наиболее удобно. Например, пять-шесть лет назад, когда железо было не таким производительным, мы все запускали cron'ы по обработке статистики по ночам. В дальнейшем мы будем говорить в том числе и об инструментах, которые используются для реализации масштабирования во времени, например об очередях.

Масштабирование “во времени”

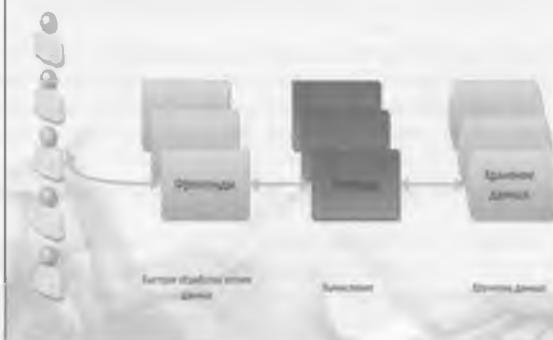
Различные данные имеют различные требования к обновлению. Это позволяет нам отложить часть обработки данных до более удобного случая.

Трехзвенная структура

Чтобы говорить на одном языке, приведем еще одно определение — определение так называемой трехзвенной структуры системы. Три звена — это фронтенд, бэкенд и хранение данных.

Каждое звено выполняет свои функции, отвечает за различные стадии в обработке запросов и по-разному масштабируется.

Трехзвенная структура



Первоначально запрос пользователя приходит на фронтенд. Фронтенды отвечают, как правило, за отдачу статических файлов, первичную обработку запроса и передачу его дальше (своему апстриму, бэкенду). Второе звено, куда приходит запрос (уже предварительно обработанный фронтенном), — это бэкенд. Бэкенд занимается вычислениями — именно он отвечает за то, чтобы вычислить, обработать, переработать, повернуть, перевернуть, перекрутить, смасштабировать и так далее. На стороне бэкенда, как правило, реализуется бизнес-логика проекта.

Следующий слой, который вступает в дело обработки запроса, — это хранение данных, которые обрабатываются бэкендом. Это может быть база данных, файловая система, да и вообще что угодно. В наших статьях мы планируем подробно описать, как масштабируется каждое из этих звеньев. Начнем с фронтенда. Для чего он нужен, мы расскажем в следующем номере. :)

Урок второй.

Масштабирование фронтендов

Напомним, на чем мы остановились в прошлый раз. При обработке запросов пользователя и обработке данных на стороне сервера выполняются операции, которые условно можно отнести к трем группам:

- предварительная обработка запроса,
- основные вычисления,
- хранение данных.

В трехзвенной архитектуре за каждое из этих действий отвечает отдельное звено. Предварительную обработку данных обеспечивает фронтенд, основные вычисления — бэкенд, хранение данных — база данных, файловая система, сетевое хранилище или что-то еще.



Фронтенд — первое звено на серверной стороне, которое и начинает обработку запроса. Зачем нужны фронтенды? Как правило, это легкие и быстрые веб-серверы, практически не занимающиеся вычислениями. Программное обеспечение фронтенда принимает запрос; далее если может, то сразу отвечает на него или, если не может, проксирует запрос к бэкенду.

Какие запросы обрабатывает фронтенд и почему?

Обычно фронтенд представляет собой легковесный веб-сервер, разработчики которого сделали все для того, чтобы каждый запрос обрабатывался максимально быстро при минимальных затратах ресурсов. Например, у nginx на 10 тысяч неактивных keep-alive-соединений уходит не более 2,5 мегабайт памяти. В правильных веб-серверах даже файлы с дисков отдаются сразу в память, минуя загрузку (такого эффекта можно достичь, включив, например, опцию `sendfile` в nginx).

Так как фронтенд (в каноническом понимании) не обрабатывает данные, то ему и не нужно большое количество ресурсов на обработку запроса. Однако тяжеловесные PHP- или Perl-процессы с многочисленными загруженными модулями могут требовать по несколько десятков мегабайт на соединение. При разработке самой первой версии nginx шла настоящая борьба за каждый килобайт, выделяемый на обработку запроса. Благодаря этому nginx тратит на обработку запроса около 8–10 килобайт, в то время как `mod_perl` может распухнуть до 200 мегабайт. Это означает, что на машинке с 16 гигабайтами оперативки удастся запустить всего лишь 40 `mod_perl`'ов, однако та же самая машинка сможет обрабатывать несколько тысяч легких соединений.

Отдача статики

Правило простое: там, где не нужно отправлять запрос на бэкенд, где не нужно что-либо вычислять (очевидно, что существует класс запросов, для обработки которых это не требуется), все должно отдаваться фронтеном. Отсюда следует первое применение фронтенда — отдача дизайнерской статики, картинок, CSS-файлов, то есть всего, что не требует вычислений. В конфигурационном файле nginx (одно из наиболее удачных решений для фронтенда) вы прописываете, какие именно запросы должны отдаваться с локального диска, а какие передаваться дальше. Наличие фронтендов — это первый признак высоконагруженной системы.

Почему это не просто важно, а очень важно? Посмотрите на любую страницу, например в Facebook. Попробуйте посчитать количество картинок на ней, затем подключаемых CSS- и JavaScript-файлов — счет пойдет на сотни. Если каждый из этих запросов отправлять бэкенду, то никакой памяти и производительности серверов не хватит. Используя фронтенд, мы сокращаем требуемые для обработки запроса ресурсы, причем зачастую в десятки и сотни раз.

Для чего нужен фронтенд?

- Отдача статического контента;
- Буферизация запросов;
- Масштабирование бекендов;
- Обслуживание медленных клиентов.

Как вариант, фронтенд может отвечать за отдачу хранящихся на диске бинарных данных пользователей. В этом случае бэкенд также не участвует в обработке запроса. Если бэкенд отсутствует, фронтенд напрямую обращается к хранилищу данных.

Отдача бинарных данных без бекенда



В качестве примера рассмотрим хранилище видеофайлов пользователей, которое размещено на десяти серверах с большими быстрыми дисками. Запрос на видеофайл пользователя приходит на фронтенд, где nginx (или любая другая аналогичная программа) определяет (например, по URI или по имени пользователя), на каком из десяти серверов лежит требуемый файл. Затем запрос отправляется напрямую на этот сервер, где другой, локальный, nginx выдает искомый файл с локального диска.

В крупных системах таких цепочек nginx'ов или подобных быстрых систем может быть довольно много.

Кеширование

Кеширование — вторая сфера применения фронтенда, некогда очень и очень популярная. В качестве грубого решения можно просто закешировать на некоторое время ответ от бэкенда.

Nginx научился кешировать относительно недавно. Он кеширует ответы от бэкенда в файлы, при этом вы можете настроить и ключ для кеширования (включив в него, например, куки пользователя), и множество других параметров для тонкого тюнинга процесса кеширования.

Соответствующие модули есть у большинства легких веб-серверов. В качестве ключа в этих модулях применяется, как правило, смесь URI- и GET-параметров.

Отдельно стоит упомянуть о потенциальных проблемах кеширования на фронтенде. Одна из них — одновременная попытка вычислить просроченное значение кеша популярной страницы. Если вы кешируете главную страницу, то при сбрасывании ее значения вы можете получить сразу несколько запросов к бэкенду на вычисление этой страницы.

В nginx имеется два механизма для решения подобных проблем. Первый механизм построен на директиве proxy_cache_lock. При ее использовании только первый запрос вычисляет новое значение элемента кеша. Все остальные запросы этого элемента ожидают появления ответа в кеше или истечения тайм-аута. Второй механизм — мягкое устаревание кеша, когда при определенных настройках, заданных с помощью директив, пользователю отдается уже устаревшее значение.

Строго говоря, кеширование на фронтенде — довольно сомнительный прием, ведь вы лишаетесь контроля над целостностью кеша. Вы обновляете страницу, но фронтенд об этом не знает и продолжает отдавать закешированную устаревшую информацию.

Вычислительная логика на стороне клиента

На стороне клиента теперь выполняется огромное количество JavaScript-кода, это способ «размазать» вычислительную логику. Фронтенд отдает браузеру клиента статику, и на стороне клиента проводятся какие-то вычисления — одна часть вычислений. А на стороне бэкенда выполняются более сложные задачи — это вторая часть вычислений.

Для примера приведу все тот же Facebook. При просмотре новостной ленты выполняется огромное количество кода на JavaScript. В «маленьком» браузере работает довольно серьезная «машинка», которая умеет очень многое. Страница Facebook загружается в несколько потоков и постоянно продолжает обновляться. За всем этим следит JavaScript, работающий у вас в браузере. Если вспомнить первый урок, то мы говорили о монолитной архитектуре. Так вот, использовать ее сейчас зачастую невозможно, поскольку приложения выполняются много где: и на стороне клиента, и на стороне сервера и так далее.

Однако попытки создать монолитные приложения, «размазанные» между браузером и сервером, все же предпринимались. Так, в качестве инструмента для написания приложений на Java-сервере, позволяющего прозрачно переносить их на клиентскую сторону, был разработан GWT (Google Web Toolkit).

Сюда также относятся всякие штуки от Microsoft типа Web Forms, которые якобы должны сами генерировать на JavaScript все, что нужно. Тем не менее про все эти решения можно сказать одно: они работают очень мучительно. На данный момент практически не существует легких в использовании и хороших средств, которые волшебным образом избавляли бы от необходимости писать отдельное приложение на JavaScript.

Масштабирование бэкендов

Одна из основных функций фронтенда — балансировка нагрузки между бэкендами, точнее, не столько балансировка, сколько проксирование.

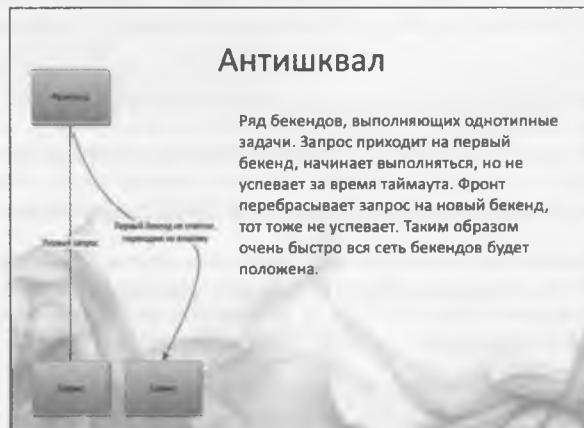
Огромный сайт «ВКонтакте» взаимодействует с внешним миром с помощью 30–40 фронтендов, за которыми скрываются многие тысячи бэкендов, выполняющих вычисления. В настройках фронтендов прописываются так называемые апстримы (upstreams), то есть серверы, куда следует отправлять тот или иной запрос.

Правила для роутинга запросов довольно много. Эти правила позволяют организовать довольно сложную логику перебрасывания запросов. Например, запросы с URI /messages/ отправляются на обработку в кластер серверов для работы с сообщениями, а /photo/ — на фотохостинг и так далее, причем все эти запросы минуют вычисляющие бэкенды.

Иногда встречаются и умные фронтенды, которые учитывают текущую загруженность бэкендов при проксировании запросов, например, выбирая для проксирования наименее загруженный бэкенд. Некоторые фронтенд-серверы умеют перезапрашивать другой бэкенд, если первый не смог обработать запрос.

При использовании этих функций стоит учитывать проблему антишквала. В чем она состоит?

Проблема с антишквалом



Допустим, есть ряд бэкендов, выполняющих однотипные задачи. Запрос приходит на первый бэкенд, начинает выполняться, но не успевает до окончания тайм-аута. Умный фронтенд перебрасывает запрос на новый бэкенд, тот тоже не успевает. Таким образом, очень быстро вся сеть бэкендов ляжет.

Варианты решения:

- I. Промежуточное звено с очередью, из которого бэкенды сами забирают задачи. Проблемы этого варианта:

1. Смешение подходов — использование асинхронных методов для решения синхронной задачи.
2. Дальнейшее выполнение запроса, когда фронтенд отключился и больше не ждет ответа.
3. Исчезновение задач, которые попали на тормозящий бэкенд (это решается рестартом очереди).



II. Умные запросы от фронтенда:

1. Первый запрос к первому бэкенду идет с тайм-аутом в одну секунду. Второй запрос идет с тайм-аутом две секунды, третий — три секунды, а четвертого уже нет, то есть мы ограничиваем количество запросов.
2. Бэкенд может определять, не перегружен ли он (раз в секунду спрашивать LA и кешировать его). В начале обработки запроса выполняется проверка. Если LA слишком высокий, фронтенду отдается Gone Away (штатная ситуация — переход к другому бэкенду).



В любом случае бэкенд получает информацию о том, сколько времени ее ответ будет ждать фронтенд, сколько времени запрос будет актуален.

Медленные клиенты

Перейдем теперь к еще одной из основных сфер применения фронтендов и поговорим о так называемом обслуживании медленных клиентов.

Представьте, что вы заходите на страницу, например, РБК (rbc.ru) и начинаете ее загружать. Страницы у них по одному-два мегабайта. Соединение не очень хорошее — вы на конференции, в роуминге, используете GPRS, — и вот эта страница загружается, загружается, загружается... Раньше такое было повсеместным и сейчас тоже случается, хотя и гораздо реже.

Рассмотрим, как происходит обработка запроса в nginx. Браузер клиента открывает соединение с одним из процессов nginx'a. Затем клиент передает этому процессу данные запроса. Одновременно процесс nginx может обрабатывать еще тысячи других соединений. Для каждого соединения существует свой входной буфер, в который закачивается запрос пользователя.

Только полностью записав запрос в буфер, nginx открывает соединение с противоположной стороной — бэкендом — и начинает проксировать запрос ему. (Если запрос очень большой, то данные в отведенную под буфер память не поместятся и nginx запишет их на диск — это один из параметров для тюнинга nginx.)

Этот же механизм действует и в обратном направлении — фронтенд буферизует ответ, полученный от бэкенда, и потихоньку отдает клиенту.

Если бы пользователь напрямую общался с процессом бэкенда, процесс бы вычислил ответ, причем моментально, за десятую долю секунды, а потом ждал, пока пользователь скушает его по одному килобайту. Все это время процесс бэкенда был бы занят и не принимал бы других запросов.

В том числе и для решения этой задачи устанавливаются легкие фронтенды. Таких фронтендов много, необязательно использовать nginx. Для бэкенда фронтенд выглядит как обычный очень быстрый браузер. Он очень быстро получает ответ от бэкенда, сохраняет этот ответ и потихоньку отдает конечному пользователю, то есть решает пресловутую проблему последней мили. Держать две минуты соединение на фронтенде — это гораздо дешевле, чем держать процесс на бэкенде.

Таким образом, мы описали основные задачи, которые решает фронтенд.

Масштабирование фронтендов

Одним из важнейших условий того, чтобы все работало и проект можно было масштабировать горизонтально, является возможность поставить дополнительные сервера. Обеспечить эту возможность непросто. Каким-то образом вы должны выставить в интернет большое количество серверов и направить пользователей на те из них, которые работают.

Кроме того, увеличение количества серверов вызывает и другие трудности. Допустим, один сервер выходит из строя раз в год. Но при наличии двух серверов сбои будут возникать раз в полгода. Если серверов уже тысяча, неисправности случаются постоянно. На каждом этапе нужно обеспечивать бесперебойную работу системы, когда ломается одна из множества одинаковых деталей.

Когда запрос уже попал в вашу систему (мы говорим про бэкенды и прочее), тут уже вы вольны программировать, как хотите. Но до того, как запрос попадает от фронтенда к бэкенду, он сначала должен попасть на фронтенд. Браузер пользователя должен к какому-то компьютеру послать какие-то данные. Отдельная сложная задача — сделать так, чтобы это было хорошо, просто и надежно.

Она имеет два аспекта. Первый из них — это технология. Раньше, например в 2001 году, технология балансировки была реализована элементарно. Когда вы заходили на spylog.ru, DNS в зависимости от того, откуда вы и кто вы, выдавал вам www1.spylog.ru, или www2.spylog.ru, или www3.spylog.ru. Сегодня большинство веб-сайтов давно уже не прибегает к этому способу. Они используют либо IPVS, либо NAT.

Таким образом, один аспект задачи состоит в том, как послать данные на работающую машину.



Второй аспект заключается в том, как понять, какая машина работает. Для этого необходим мониторинг. В простейшем случае этот мониторинг представляет собой проверку того, отвечает ли машина на ping. При более глубоком рассмотрении оказывается, что сама эта проблема разделяется на несколько других.

Вы начинаете мониторинг. Допустим, вы обнаруживаете, что у машины живая сетевая карта, но сгорел диск. То есть вы хотите сделать балансировку, распределить нагрузку, а машина банально тормозит. Мониторинг и роутинг как раз и позволяют решить задачу балансировки.

DNS-балансировка

Вернемся к первому аспекту — к отправке запроса на ваши фронтенды. Самый простой способ, который используется до сих пор, — это DNS-балансировка, то есть эти несколько машин, куда нужно отправлять пользователя, зашиты в DNS.

Начинать проще всего с TTL в пять или в одну минуту (то есть с минимального, который разумно выставить). Пока у вас три-пять фронтендов, что, на самом деле, тоже немало, это на довольно долгое время убережет вас от проблем. Когда же их больше...

Вы, конечно, можете возразить, что часть провайдеров любит кэшировать. В этом случае TTL длительностью пять минут превращается в проблему.

Однако трудности возникнут в любом случае, какое бы решение вы ни выбрали. Если вы, например, от DNS-балансировки перешли к выделенной железке, к IPVS, появятся проблемы с нагрузкой этой железки. Они также могут быть связаны с надежностью дата-центра или дистрибуцией контента. Тут очень много аспектов.

Из всего вышесказанного можно вывести правило, которое применимо при разработке любой крупной системы, — решаем проблемы по мере их появления, каждый раз выбирая наиболее простое решение из всех возможных.

Отказоустойчивость фронтенда

Рассмотрим чуть более сложный способ, который часто используется и имеет кучу вариантов. Как он реализуется? Ставим рядом две машинки, у каждой из которых две сетевых карты. С помощью одной каждой из них «смотрит в мир», с помощью другой они слушают и мониторят друг с друга. Внешние сетевые карты имеют одинаковые IP-адреса. Весь поток идет через первую машину. Как только одна из них умирает, поднимается IP-адрес на второй. Именно так реализованы CARP (во FreeBSD), Heartbeat (в Linux) и другие соединения подобного рода.



Такая схема долгое время работала в Rambler, и, насколько я понимаю, она используется повсеместно. У вас есть DNS-балансировка, разбрасывающая пользователей на пары серверов, в каждой из которых серверы контролируют друг друга.

Перейдем к балансировке бэкендов. Она осуществляется на уровне фронтенда. У него есть простой сервис, который знает все свои так называемые upstream'ы и логику, по которой между ними разбрасываются запросы. В подавляющем большинстве случаев это происходит случайным образом. Но можно задать какие-то обратные связи, посыпать запрос не на ближайший upstream, а на тот, который меньше всего нагружен, и так далее.

Речь идет о том, что для множества сайтов хватает всего двух фронтендов, причем с избытком. Nginx — это очень быстрая штука. А бэкенды, которые вы пишете, — это ваша бизнес-логика, и она может работать сколь угодно оптимально или неоптимально. Их, как правило, гораздо больше.

Обычный масштаб чаще всего предполагает наличие двух или четырех фронтендов и двадцати бэкендов. При этом вопрос о том, как отправить запрос тому бэкенду, который лучше всего его обслужит, остается по-прежнему актуальным.



Пожалуй, на этом о масштабировании фронтендов всё. В следующем уроке мы поговорим о том, как масштабировать бэкенды.

Стрелочки на рисунке направлены от фронтендов к бэкендам. Этот рисунок иллюстрирует интересную технологию Mongrel2, используемую в мире Ruby. Ее разработкой занимается известный в Ruby-сообществе Зед Шоу, который и предложил перевернуть привычную схему обработки запросов с ног на голову.

Согласно его схеме, не фронтенды ходят к бэкендам и предлагают им обработать какой-то запрос, а наоборот. Фронтенды накапливают у себя очередь на запросы, а огромное количество бэкендов эти фронтенды опрашивает: «Дай чего-нибудь обработать, дай на подумать» — и возвращает ответ. Таким образом, мы получаем масштабируемую асинхронную обработку.

Nginx [engine x], написанный Игорем Сысоевым, объединяет в себе HTTP-сервер, обратный прокси-сервер и почтовый прокси-сервер. Уже длительное время nginx обслуживает серверы многих высоконагруженных российских сайтов, таких как «Яндекс», Mail.Ru, «ВКонтакте» и «Рамблер». Согласно статистике Netcraft, в июне 2012 года nginx обслуживал или проксировал 10,29% самых нагруженных сайтов.

Урок 3.

Масштабирование бекенда

- 5.1. Функциональное разделение
- 5.2. Классическое горизонтальное масштабирование
 - 5.2.1. Низкая степень связности кода
 - 5.2.2. Share-nothing для горизонтального масштабирования: плюсы и минусы
 - 5.2.3. Слоистость кода
 - 5.2.4. Минимизация использования сложных запросов сразу к нескольким таблицам
- 5.3. Кеширование
 - 5.3.1. Общие рекомендации при использовании кеширования
 - 5.3.2. Проблема инвалидации кеша
 - 5.3.3. Проблема старта с непрогретым кешем

Начнем наш третий урок, посвященный бизнес-логике проекта. Это самая главная составляющая в обработке любого запроса. Для таких вычислений требуются бэкенды — тяжелые серверы с большими вычислительными мощностями. Если фронтенд не может отдать клиенту что-то самостоятельно (а как мы выяснили в прошлом номере, он без проблем можем сам отдать, к примеру, картинки). На бэкенде отрабатывается бизнес-логика, то есть формируются и обрабатываются данные, при этом данные хранятся в другом слое — сетевом хранилище, базе данных или файловой системе. Хранение данных — это тема следующего урока, а сегодня мы сосредоточимся на масштабировании бекенда.

Сразу предупредим: масштабирование вычисляющих бэкендов — одна из самых сложных тем, в которой существует множество мифов. Облачные вычисления решают проблему производительности — уверены многие. Однако это верно не до конца: для того чтобы вам действительно могли помочь облачные сервисы, вы должны правильно подготовить ваш программный код. Вы можете поднять сколько угодно серверов, скажем, в Amazon EC2, но какой с них толк, если код не умеет использовать мощности каждого из них. Итак, как масштабировать бэкенд?

Функциональное разделение

Самый первый и простой способ, с которым сталкиваются все, — это функциональное разбиение, при котором разные части системы, каждая из которых решает строго свою задачу, разносятся на отдельные физические серверы. Например, посещаемый форум выносится на один сервер, а все остальное работает на другом.



Несмотря на простоту, о подобном подходе многие забывают. Например, мы очень часто встречаем веб-проекты, где используется только одна база MySQL под совершенно различные типы данных. В одной базе лежат и статьи, и баннеры, и статистика, хотя по-хорошему это должны быть разные экземпляры MySQL. Если у вас есть функционально не связанные данные (как в этом примере), то их целесообразно разносить в разные экземпляры баз данных или даже физические серверы. Посмотрим на это с другой стороны. Если у вас есть в одном проекте истроенная интегрированная баннерокрутка, и сервис, который показывает посты пользователей, то разумное решение — сразу осознать, что эти данные никак не связаны между собой и поэтому должны жить в самом простом варианте в двух разных запущенных MySQL. Это относится и к вычисляющим бэкендам — они тоже могут быть разными. С совершенно разными настройками, с разными используемыми технологиями и написанные на разных языках программирования. Возвращаясь к примеру: для показа постов вы можете использовать в качестве бэкенда самый обычный PHP, а для баннерной системы вы можете запустить модуль к nginx'у. Соответственно, для постов вы можете выделить сервер с большим количеством памяти (ну PHP все-таки), при этом для баннерной системы память может быть не так важна, как процессорная емкость.

Сделаем выводы: функциональное разбиение бэкенда целесообразно использовать в качестве простейшего метода масштабирования. Группируйте сходные функции и запускайте их обработчики на разных физических серверах.

Обратимся к следующему подходу.

Классическое горизонтальное масштабирование

О том, что такое горизонтальное масштабирование, в принципе, мы уже знаем. Если вашей системе не хватает мощности, вы просто добавляете еще десять серверов, и они продолжают работать. Но не каждый проект позволит провернуть такое. Есть несколько классических парадигм, которые необходимо рассмотреть на раннем этапе проектирования, чтобы программный код можно было масштабировать при росте нагрузки.

Классическое горизонтальное масштабирование

- Shared nothing (каждый узел является независимым и самодостаточным, не существует единой точки отказа).
- Stateless (процесс не хранит состояние).

Концепции Shared Nothing и Stateless

Рассмотрим две концепции — Shared Nothing и Stateless, которые могут обеспечить возможность горизонтального масштабирования.

Подход Shared Nothing означает, что каждый узел является независимым, самодостаточным и нет какой-то единой точки отказа. Это, конечно, не всегда возможно, но в любом случае количество таких точек находится под жестким контролем архитектора. Под точкой отказа мы понимаем некие данные или вычисления, которые являются общими для всех бэкендов.

Например, какой-нибудь диспетчер состояний или идентификаторов. Другой пример — использование сетевых файловых систем. Это прямой путь получить на определенном этапе роста проекта узкое место в архитектуре. Если каждый узел является независимым, то мы легко можем добавить еще несколько — по росту нагрузки.

Концепция Stateless означает, что процесс программы не хранит свое состояние. Пользователь пришел и попал на этот конкретный сервер, и нет никакой разницы, попал пользователь на этот сервер или на другой. После того как запрос будет обработан, этот сервер полностью забудет информацию об этом пользователе. Пользователь вовсе не обязан все свои следующие запросы отправлять на этот же сервер, не должен второй раз приходить на него же. Таким образом, мы можем динамически менять количество серверов и не заботиться о том, чтобы роутить пользователя на нужный сервер.

Наверное, это одна из серьезных причин, почему веб так быстро развивается. В нем гораздо проще делать приложения, чем писать классические онлайновые программы. Концепция «ответ — запрос» и тот факт, что ваша программа живет 200 миллисекунд или максимум одну секунду (после чего она полностью уничтожается), привели к тому, что в таких распространенных языках программирования, как PHP, до сих пор нет сборщика мусора.

Описанный подход является классическим: он простой и надежный, как скала. Однако в последнее время нам все чаще и чаще приходится отказываться от него.

Критика концепций Shared Nothing и Stateless

Сегодня перед вебом возникают новые задачи, которые ставят новые проблемы. Когда мы говорим про Stateless, это означает, что каждые данные каждому пользователю мы заново тащим из хранилища, а это подчас бывает очень дорого. Возникает резонное желание положить какие-то данные в память, сделать не совсем Stateless. Это связано с тем, что сегодня веб становится все более и более интерактивным. Если вчера человек заходил в веб-почту и нажимал на кнопку «Reload», чтобы проверить новые сообщения, то сегодня этим уже занимается сервер. Он ему говорит: «О, чувак, пока ты сидел на этой страничке, тебе пришли новые сообщения».

Возникают новые задачи, которые приводят к тому, что подход с Shared Nothing и отсутствием состояния в памяти иногда не является обязательным. Мы уже сталкивались неоднократно с ситуациями наших клиентов, которым мы говорим: «От этого откажитесь, положите данные в память» и наоборот «Направляйте людей на один и тот же сервер». Например, когда возникает открытая чат-комната, людей имеет смысл роутить на один и тот же сервер, чтобы это все работало быстрее.

Расскажем про еще один случай, с которым сталкивались. Один наш знакомый разрабатывал на Ruby on Rails игрушку наподобие «Арены» (онлайн драки и бои). Вскоре после запуска он столкнулся с классической проблемой: если несколько человек находятся в рамках одного боя, каждый пользователь постоянно вытаскивает из БД данные, которые во время этого боя возникли. В итоге вся эта конструкция смогла дожить только до 30 тысяч зарегистрированных юзеров, а дальше она просто перестала работать.

Обратная ситуация сложилась у компании Vuga, которая занимается играми для Facebook. Правда, когда они столкнулись с похожей проблемой, у них были другие масштабы: несколько миллиардов SELECT'ов из PostgreSQL в день на одной системе. Они перешли полностью на подход Memory State: данные начали храниться и обслуживаться прямо в оперативной памяти. Итог: ребята практически отказались от базы данных, а пара сотен серверов оказались лишним. Их просто выключили: они стали не нужны.

В принципе, любое масштабирование (в том числе горизонтальное) достижимо на очень многих технологиях. Сейчас очень часто речь идет о том, чтобы при создании сервиса не пришлось платить слишком много за железо. Для этого важно знать, какая технология наибо-

лее соответствует данному профилю нагрузки с минимальными затратами железа. При этом очень часто, когда начинают размышлять о масштабировании, то забывают про финансовый аспект того же горизонтального масштабирования. Некоторые думают, что горизонтальное масштабирование — это реально панацея. Разнесли данные, все разбросали на отдельные серверы — и все стало нормально. Однако эти люди забывают о накладных расходах (оверхедах) — как финансовых (покупка новых серверов), так эксплуатационных. Когда мы разносим все на компоненты, возникают накладные расходы на коммуникацию программных компонентов между собой. Грубо говоря, хопов становится больше. Вспомним уже знакомый тебе пример. Когда мы заходим на страничку Facebook, мощный JavaScript идет на сервер, который долго-долго думает и только через некоторое время начинает отдавать вам ваши данные. Все наблюдали подобную картину: хочется уже посмотреть и бежать дальше пить кофе, а оно все грузится, грузится и грузится. Надо бы хранить данные чуть-чуть « pobliже», но у Facebook уже такой возможности нет.

Слоистость кода

Классическое горизонтальное масштабирование

- Слоистость кода;
- Минимизация использования сложных запросов сразу к нескольким таблицам;
- Низкая степень связности кода;

Еще пара советов для упрощения горизонтального масштабирования. Первая рекомендация: программируйте так, чтобы ваш код состоял как бы из слоев и каждый слой отвечал за какой-то определенный процесс в цепочке обработки данных. Скажем, если у вас идет работа с базой данных, то она должна осуществляться в одном месте, а не быть разбросанной по всем скриптам. К примеру, мы строим страницу пользователя. Все начинается с того, что ядро запускает модуль бизнес-логики для построения страницы пользователя. Этот модуль запрашивает у нижележащего слоя хранения данных информацию об этом конкретном пользователе. Слою бизнес-логики ничего не известно о том, где лежат данные: закешированы ли они, зашардированы ли (шардинг — это разнесение данных на разные серверы хранения данных, о чем мы будем говорить в будущих уроках), или с ними сделали еще что-нибудь нехорошее. Модуль просто запрашивает информацию, вызывая соответствующую функцию. Функция чтения информации о пользователе расположена в слое хранения данных. В свою очередь, слой хранения данных по типу запроса определяет, в каком именно хранилище хранится пользователь. В кеше? В базе данных? В файловой системе? И далее вызывает соответствующую функцию нижележащего слоя.

Что дает такая слоистая схема? Она дает возможность переписывать, выкидывать или добавлять целые слои. Например, решили вы добавить кеширование для пользователей. Сделать это в слоистой схеме очень просто: надо допилить только одно место — слой хранения данных. Или вы добавляете шардирование, и теперь пользователи могут лежать в разных базах данных. В обычной схеме вам придется перелопатить весь сайт и везде вставить соответствующие проверки. В слоистой схеме нужно лишь исправить логику одного слоя, одного конкретного модуля.

Связность кода и данных

Следующая важная задача, которую необходимо решить, чтобы избежать проблем при горизонтальном масштабировании, — минимизировать связность как кода, так и данных. Например, если у вас в SQL-запросах используются JOIN'ы, у вас уже есть потенциальная проблема. Сделать JOIN в рамках одной базы данных можно. А в рамках двух баз данных, разнесенных по разным серверам, уже невозможно. Общая рекомендация: старайтесь общаться с хранилищем минимально простыми запросами, итерациями, шагами.

Что делать, если без JOIN'a не обойтись? Сделайте его сами: сделали два запроса, перемножили в PHP — в этом нет ничего страшного. Для примера рассмотрим классическую задачу построения френдленты. Вам нужно поднять всех друзей пользователя, для них запросить все последние записи, для всех записей собрать количество комментариев — вот где соблазн сделать это одним запросом (с некоторым количеством вложенных JOIN'ов) особенно велик. Всего один запрос — и вы получаете всю нужную вам информацию. Но что вы будете делать, когда пользователей и записей станет много и база данных перестанет справляться? По-хорошему надо бы расшардить пользователей (разнести равномерно на разные серверы баз данных). Понятно, что в этом случае выполнить операцию JOIN уже не получится: данные-то разделены по разным базам. Так что придется делать все вручную. Вывод очевиден: делайте это вручную с самого начала. Сначала запросите из базы данных всех друзей пользователя (первый запрос). Затем заберите последние записи этих пользователей (второй запрос или группа запросов). Затем в памяти произведите сортировку и выберите то, что вам нужно. Фактически вы выполняете операцию JOIN вручную. Да, возможно вы выполните ее не так эффективно, как это сделала бы база данных. Но зато вы никак не ограничены объемом этой базы данных в хранении информации. Вы можете разделять и разносить ваши данные на разные серверы или даже в разные СУБД! Все это совсем не так страшно, как может показаться. В правильно построенной слоистой системе большая часть этих запросов будет закеширована. Они простые и легко кешируются — в отличие от результатов выполнения операции JOIN. Еще один минус варианта с JOIN: при добавлении пользователем новой записи вам нужно сбросить кэши выборок всех его друзей! А при таком раскладе неизвестно, что на самом деле будет работать быстрее.

Кеширование

Следующий важный инструмент, с которым мы сегодня познакомимся, — кеширование. Что такое кеш? Кеш — это такое место, куда можно под каким-то ключом положить данные, которые долго вычисляют. Запомните один из ключевых моментов: кеш должен вам по этому ключу отдать данные быстрее, чем вычислить их заново. Мы неоднократно сталкивались с ситуацией, когда это было не так и люди бессмысленно теряли время. Иногда база данных работает достаточно быстро и проще сходить напрямую к ней. Второй ключевой момент: кеш должен быть единственным для всех бекендов.



Второй важный момент. Кеш — это скорее способ замазать проблему производительности, а не решить ее. Но, безусловно, бывают ситуации, когда решить проблему очень дорого. Поэтому вы говорите: «Хорошо, эту трещину в стене я замажу штукатуркой, и будем думать, что ее здесь нет». Иногда это работает — более того, это работает очень даже часто. Особенно когда вы попадаете в кэш и там уже лежат данные, которые вы хотели показать. Классический пример — счетчик количества друзей. Это счетчик в базе данных, и вместо того, чтобы перебирать всю базу данных в поисках ваших друзей, гораздо проще эти данные закешировать (и не пересчитывать каждый раз).

Для кеша есть критерий эффективности использования, то есть показатель того, что он работает, — он называется Hit Ratio. Это отношение количества запросов, для которых ответ нашелся в кеше, к общему числу запросов. Если он низкий (50–60%), значит, у вас есть лишние накладные расходы на поход к кешу. Это означает, что практически на каждой второй странице пользователь, вместо того чтобы получить данные из базы, еще и ходит к кешу: выясняет, что данных для него там нет, после чего идет напрямую к базе. А это лишние две, пять, десять, сорок миллисекунд.

Как обеспечивать хорошее Hit Ratio? В тех местах, где у вас база данных тормозит, и в тех местах, где данные можно перевычислять достаточно долго, там вы втыкаете Memcache, Redis или аналогичный инструмент, который будет выполнять функцию быстрого кеша, — и это начинает вас спасать. По крайней мере, временно.

Проблема инвалидации кеша

Но с использованием кеша вы бонусом получаете проблему инвалидации кеша. В чем суть? Вы положили данные в кэш и берете их из кеша, однако к этому моменту оригинальные данные уже поменялись. Например, Машенька поменяла подпись под своей картинкой, а вы зачем-то положили одну строчку в кэш вместо того, чтобы тянуть каждый раз из базы данных. В результате вы показываете старые данные — это и есть проблема инвалидации кеша. В общем случае она не имеет решения, потому что эта проблема связана с использованием данных вашего бизнес-приложения. Основной вопрос: когда обновлять кеш? Ответить на него подчас непросто. Например, пользователь публикует в социальной сети новый пост — допустим, в этот момент мы пытаемся избавиться от всех инвалидных данных. Получается, нужно сбросить и обновить все кеши, которые имеют отношение к этому посту. В худшем случае, если человек делает пост, вы сбрасываете кэш с его ленты постов, сбрасываете все кеши с ленты постов его друзей, сбрасываете все кеши с ленты людей, у которых в друзьях есть те, кто в этом сообществе, и так далее. В итоге вы сбрасываете половину кешей в системе. Когда Цукерберг публикует пост для своих одиннадцати с половиной миллионов подписчиков, мы что — должны сбросить одиннадцать с половиной миллионов кешей френдлент у всех этих subscriber'ов? Как быть с такой ситуацией? Нет, мы пойдем другим путем и будем обновлять кэш при запросе на френдленту, где есть этот новый пост. Система обнаруживает, что кеша нет, идет и вычисляет заново. Подход простой и надежный, как скала. Однако есть и минусы: если сбросился кэш у популярной страницы, вы рискуете получить так называемые race-condition (состояние гонок), то есть ситуацию, когда этот самый кэш будет одновременно вычисляться несколькими процессами (несколько пользователей решили обратиться к новым данным). В итоге ваша система занимается довольно пустой деятельностью — одновременным вычислением п-го количества одинаковых данных.



Один из выходов — одновременное использование нескольких подходов. Вы не просто стираете устаревшее значение из кеша, а только помечаете его как устаревшее и одновременно ставите задачу в очередь на пересчет нового значения. Пока задание в очереди обрабатывается, пользователю отдается устаревшее значение. Это называется деградацией функциональности: вы сознательно идете на то, что некоторые из пользователей получат не самые свежие данные. Большинство систем с продуманной бизнес-логикой имеют в арсенале подобный подход.

Проблема старта с непрогретым кешем

Еще одна проблема — старт с непрогретым (то есть незаполненным) кешем. Такая ситуация наглядно иллюстрирует утверждение о том, что кэш не может решить проблему медленной базы данных. Предположим, что вам нужно показать пользователям 20 самых хороших постов за какой-либо период. Эта информация была у вас в кеше, но к моменту запуска системы кэш был очищен. Соответственно, все пользователи обращаются к базе данных, которой для построения индекса нужно, скажем, 500 миллисекунд. В итоге все начинает медленно работать, и вы сами себе сделали DoS (Denial-of-service). Сайт не работает. Отсюда вывод: не занимайтесь кешированием, пока у вас не решены другие проблемы. Сделайте, чтобы база быстро работала, и вам не нужно будет вообще возиться с кешированием. Тем не менее даже у проблемы старта с незаполненным кешем есть решения:

1. Использовать кеш-хранилище с записью на диск (теряем в скорости);
2. Вручную заполнять кэш перед стартом (пользователи ждут и негодуют);
3. Пускать пользователей на сайт партиями (пользователи все так же ждут и негодуют).

Как видите, любой способ плох, поэтому лишь повторимся: старайтесь сделать так, чтобы ваша система работала и без кеширования.

Урок 4.

Масштабирование во времени

В прошлых уроках мы говорили о том, как писать программы так, чтобы их можно было запустить в нескольких экземплярах и тем самым выдерживать большую нагрузку. В этом уроке мы поговорим о еще более интересных вещах — как использовать для построения технической архитектуры знания о бизнес-логике продукта и как обрабатывать данные тогда, когда это нужно, максимально эффективно используя аппаратную инфраструктуру.

Отложенные вычисления

Когда пользователь вводит запрос на сайт, необходимо дать ему ответ, и для этого сначала придется проделать соответствующую работу. Скажем, если человек сделал модификационный запрос (например, создал новый пост), то вам предстоит провернуть огромный объем работы. Недостаточно просто положить пост. Нужно обновить счетчики, оповестить друзей, разослать электронные уведомления. Хорошая новость: делать все сразу необязательно.

Тот же самый Facebook после того, как вы публикуете новый пост, делает еще одиннадцать разных вещей — и это только то, что видно снаружи невооруженным глазом. Причем все эти операции исполняются в разное время. Например, электронное письмо с уведомлением можно послать вообще минут через десять.

Этот принцип мы и возьмем на вооружение. Любой маленький сайт, начиная рести по нагрузке, сталкивается с тем, что, оказывается, больше нельзя делать все необходимые операции синхронно в функции обработки самого модификационного запроса. В противном случае пользователь не получит моментальный ответ.

Современные языки веб-программирования часто не позволяют в явном виде реализовать такой трюк, поскольку в них не предусмотрена возможность делать что-либо после ответа. Однако есть и исключения. В популярном веб-сервере Apache существует более десяти стадий обработки запроса, включая трансляцию URI, авторизацию, аутентификацию и собственно обработку запроса. Сюда же входят и стадии, которые выполняются уже после того, как ответ пользователю отправлен. Некоторые фреймворки (например, mod_perl) позволяют перехватить эти стадии и повесить на них ваши собственные функции. Можно передать в эти функции данные для обработки и спокойно обрабатывать их уже после того, как пользователь получил ответ.

Асинхронная обработка

Однако иногда описанного выше подхода с постобработкой данных недостаточно. Действия, которые надо совершить с ними, могут занимать слишком много времени, а ресурсы веб-сервера небезграничны.

В таком случае помогает следующий архитектурный паттерн — сохраните данные в некое промежуточное хранилище, а затем обработайте их с помощью отдельного асинхронного процесса. Термин «асинхронность» означает в общем случае разнесенность во времени. То есть данные собираются сейчас, а обрабатываются тогда, когда будет удобно.

Например, очень часто асинхронно обсчитывается статистика уникальных посетителей — раз в день, как правило по ночам, в часы наименьшей загрузки, запускается скрипт, который берет весь массив данных, накопленных за день, обрабатывает их и сохраняет уже в другом виде. Такой подход применяется при разработке баннерных сетей, счетчиков и других подобных проектов. Часто в часы наименьшей нагрузки выполняются и различные обслуживающие процедуры: оптимизация баз данных, бэкапы.

Обратите внимание — мы уже используем наше знание о бизнес-процессах в проекте. Мы знаем, что детальную статистику за день пользователи готовы подождать, и учитываем этот факт при проектировании архитектуры проекта. Более углубленное использование этих знаний мы с вами разберем дальше.

Очереди

Рассмотрим подробнее инструменты, позволяющие отложить на потом те вещи, которые «не горят». Одно из уже упоминавшихся решений — промежуточное хранилище. Но существует целый класс однотипных задач, для которых хотелось бы, чтобы это хранилище обладало определенными свойствами. Это уже не просто данные для обработки — тут важен порядок, важна очередьность.

Для подобных целей используют инструмент, называемый очередями, — особый вид хранилища, поддерживающий логику FIFO (первый вошел — первый вышел). Получаются очереди сообщений и очереди задач, которые надо делать. Например, вместо того чтобы слать e-mail, можно поместить в очередь задачу: «Послать e-mail». Какой-то фоновый скрипт, который запущен, вытянет из очереди новый запрос. «О, надо послать e-mail. Сейчас пошлю его».

Очереди — довольно продвинутый и часто встречающийся инструмент. Например, даже когда пользователь в Windows кликает мышкой на кнопку в приложении, то приложение не принимается немедленно обрабатывать это событие (ведь в этот момент приложение может быть занято другим действием). Операционная система присыпает приложению сообщение, содержащее описание совершенного пользователем действия. Сообщение ставится в очередь и будет обработано в порядке поступления. В результате если вы два раза кликните мышкой на два разных пункта меню, то сначала откроется одно, потом другое.

Для использования очередь есть ряд инструментов, один из наиболее популярных сейчас — *RabbitMQ* (www.rabbitmq.com), написанный на Erlang'e. Причем несмотря на колоссальные возможности по обслуживанию очередей сообщений, начать использовать его крайне просто. Практически для любого языка программирования (PHP, Python, Ruby и так далее) есть готовые библиотеки.

В крупных веб-системах могут использоваться одновременно десятки очередей: очередь на отправку электронной почты, очередь для обновления счетчиков, очередь для обновления френдлент пользователей.

По большому счету очереди — это пример межсервисной коммуникации, когда один сервис (публикация поста) ставит задачи другому сервису (рассылка электронной почты). Рассмотрим это подробнее на конкретном примере.

Пример премодерируемой социальной сети

Рассмотрим пример большого проекта социальной сети уровня Facebook. Пользователи посты в огромном количестве, генерируются гигантские объемы различных операций. Но допустим, что это не просто Facebook, а социальная сеть с премодерацией всех сообщений. Задача стала еще сложнее — что делать?

Разработчики посмотрели: посты храним так, комментарии храним так. Всё в разных табличках, может быть даже в базах данных разного вида. Например, этих баз и табличек десять. Неужели модераторский софт должен будет ходить по десятку баз данных, вытаскивать обновления за последние пять минут из всех этих табличек? Объединять все изменения в один большой список и показывать модератору? А что, если модераторов несколько? А если данных очень много?

Вот тут на помощь и приходят очереди. Любой постинг приводит не только к добавлению сообщения в большую базу данных (где оно будет жить постоянно), но и к его попаданию в некую модераторскую систему. Взаимодействие сервисов позволяет навести тут порядок.



Нижний красный блок на слайде — это исходящий сервер очередей RabbitMQ, который получает сообщение. Этот сервер кем-то слушается с той стороны, и в результате приходящие данные перегруппировываются и уже складываются в другую базу данных или в другое место, специально предназначенное для модерации. Например, они группируются, и вместо модерации может идти, например, аналитическая система. Сам бог велел входящие данные преобразовывать, чтобы потом было удобно их обрабатывать и решать данную конкретную задачу.

Но вернемся к нашему примеру. Все эти сообщения каким-то образом просматриваются, и, так как схема базы данных заточена под модерацию, это происходит легко и просто. Нам что-то не нравится — удаляем это сообщение, и запрос точно так же уходит обратно, в другую очередь: «Это сообщение из тех, что ты мне прислал, удали». Есть такой же разборщик, который берет эту задачу с удаленным сообщением, ищет, где же оно там все-таки лежит, и удаляет.

Мы получили классический пример использования очереди.

Неконсистентность данных

Здесь возникают недостатки, характерные для любой системы, которая хранит данные в двух местах. Любая проблема с оборудованием, с выполнением этой сложной функциональности — и возникает неконсистентность данных. Например, сообщение попало в основную базу, но не было добавлено в очередь. Сообщение прошло модерацию, но оно реально удалено не было, потому что очередь, ответственная за хранение сообщений на удаление, вышла из строя.

Чтобы избежать этого, нужно писать программу таким образом, чтобы при повторном ее выполнении она доходила до конца. Этот принцип называется идемпотентностью — повторное действие не изменит наши данные, если в первый раз все было сделано правильно. Увы, это далеко не всегда можно применить.

Другое решение — логическое логирование действий. Создается некий блокнот, например файл, хранящийся на каком-то надежном носителе. Программы при выполнении оставляют там записи вида: «Я успел сделать то, то и то, но еще не успел сделать вот это» и «Я собираюсь сделать это». Если что-то пошло не так, подобный отчет позволит понять, на каком этапе произошел сбой, и исправить положение.

Алгоритм проектирования архитектуры

1. Рассмотрим решение нашей проблемы, исходя из конкретных условий бизнес-логики. Сначала составляем варианты использования проекта (Use cases), функциональное описание, в нашем случае — основные веб-сервисы. Описываем, что конкретно делает пользователь на той или иной странице.

Например, страница news feed пользователя:

- загружаются десять последних объектов-записей по времени объектов, опубликованных всеми пользователями, на которых подписан пользователь;
- для каждой из записей поднимается информация о пользователе-авторе (имя, аватар и ссылка на профиль);
- для каждой из записей на странице поднимаются три последних комментария, по каждому комментарию поднимается аватар и имя пользователя.

Отдельного упоминания заслуживает обсуждение потенциальных сценариев дальнейшего развития проекта. Например, сейчас нет обновления комментариев без перезагрузки, а потом будет — такую возможность нужно предусмотреть еще на этапе начального проектирования. Далее по этим описаниям планируются потоки данных с расчетом потенциальных объемов, требований к скорости в каждом случае. Важно также проговорить потенциальную степень деградации данных.

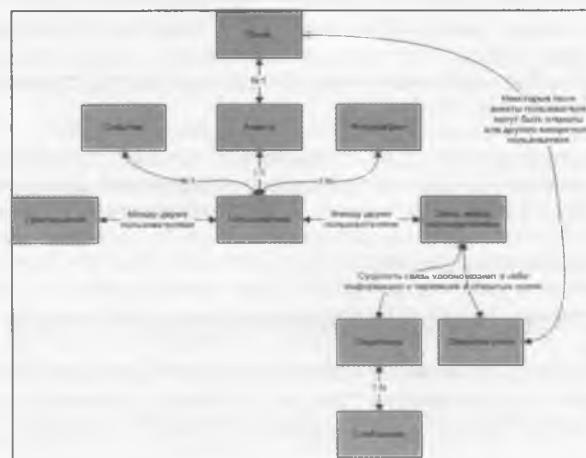
Например, у нас ожидается тридцать миллионов зарегистрированных пользователей в новой потенциальной социальной сети. По аналогии с существующими сетями предположим, что в день на сайт будет заходить зарегистрированных пользователей, то есть шесть миллионов пользователей.

Какое количество записей делает в день пользователь? Этот вопрос требует исследования, но допустим, что в среднем три сообщения в день кто-то больше, кто-то меньше. На практике большинство будет заходить на сайт только для чтения чужих сообщений, поэтому сократим в пять раз — пусть пишет по три сообщения каждый пятый пользователь.

Получается 3,6 миллиона записей в сутки. У каждого пользователя 100 подписчиков, то есть (если мы остаемся на схеме уведомлений об изменениях) 360 миллионов уведомлений в сутки.

С учетом пикового характера веб-трафика получаем 10 тысяч уведомлений в секунду — это может быть проблемой! Мы видим такую цифру и понимаем, что нам придется рассыпать уведомления не в реальном времени.

Чтобы рассчитать объемы данных, допустим, что половина сообщений текстовые, а половина — графические. Размер текстового сообщения в среднем 200 байт, графического — 100 килобайт. Итого в день мы генерируем данных на 360 мегабайт текстовых сообщений и на 180 гигабайт графики. Это немало, и очевидно, что мы не можем просто положить тексты в SQL базу данных и делать по ней выборки. Максимум — мы можем делать выборки по некоторой упрощенной информации, например таблицам с идентификаторами.



На этом этапе можно и нужно проговорить сущности и связи между ними. Например, как на приведенном выше рисунке. Это пример из реального проекта простой социальной сети, разработанной нашей компанией.

Здесь же мы проговариваем вопросы деградации, для этого продакт-менеджер должен ответить нам на следующие вопросы:

- Страшно ли, если запись друга появится в news feed пользователя не моментально, а через пять секунд? А если через десять? Через минуту? Через час? Какова допустимая задержка?
- Сколько последних записей мы выводим на одной странице? Десять? Двадцать? Могу ли я перейти к более старым записям?
- Должны ли новые записи появляться на странице без перезагрузки?
- Должны ли новые комментарии к записям, находящимся на странице, появляться без перезагрузки страницы?
- Страшно ли, если записи будут подгружаться пользователю постепенно, не сразу, сначала одна, потом еще пять, а потом вдруг раз — и загрузилась запись где-то в середине.

Здесь же мы прописываем скорость работы страницы (в нашем случае не более 0,2–0,5 секунды, например) и проговариваем какие-то особенности использования модуля. Например, в нашем случае с news feed это может быть:

- 99% пользователей просматривают ленту своих записей на одну-три страницы назад (обычно до последней прочитанной записи). Архив просматривается крайне редко. Можем ли мы как-нибудь использовать эту особенность?
- Нам не надо показывать пользователю сразу всю страницу, мы можем показать ему пару записей и, пока он смотрит на них, подгружать остальные.

Вот тут надо понаблюдать за работой конкурентов, например того же Facebook. Алгоритм работы примерно такой:

- сначала загружается обвязка;
- затем происходит запрос идентификаторов записей для данной news feed;
- затем в цикле, начиная от самых старых, запрашиваем подробности про записи.

Эту схему можно упростить — например часть данных в виде JSON загружать заранее при загрузке страницы. Если посмотреть страницы Facebook, вы увидите только JavaScript, все данные представлены в виде JSON-массивов. Это очень удобно — просто сделать AJAX-обновление страницы без перезагрузки.

2. Далее нужно сформулировать дополнительные технические требования к отказоустойчивости и скорости всей системы и отдельных веб-сервисов. Наконец, для каждого из веб-сервисов, исходя из конкретных особенностей данных и требований к каждому компоненту, проектируем архитектуру и подбираем технологии. Многие технологии имеют аналоги, и среди кластера технологий выбор стоит делать на основе предпочтений команды разработчиков. Что умеют, на том и надо работать.

Итак, у нас есть довольно существенный поток данных, обладающих, однако, определенными особенностями. Похоже, что стоит разделить сами записи и структуру news feed'ов.

Не все данные нужно показывать сразу, успокоил нас продакт-менеджер и дал пару минут, чтобы поместить новое сообщение пользователя во френдленты его друзей. Строго говоря, это может быть и не так исходя из бизнес-логики проекта. В этом случае мы с вами выбрали бы другую архитектуру для системы хранения френдлент.

При обсуждении продакт-менеджер сказал нам также, что рассылку почтовых уведомлений

мы можем отложить на потом. Мы рассчитали объем задач, прикинули систему хранения для очереди и приступили к реализации.

Строго говоря, использование серверов очередей не обязательно. Не умеете работать с RabbitMQ? Ничего страшного — используйте паттерн «Очереди», но храните список задач в обычном MySQL.

Использование очередей для достижения надежности

Усложним задачу из примера с почтовыми рассылками. Итак, воркеру, обслуживающему почтовые рассылки, нужно разослать 25 писем. Три послали, на четвертом закончилось место на диске. Хлоп, сломалось! Если вы пишете синхронную рассылку в едином PHP-коде, то у вас из-за отвалившегося почтовика может возникнуть 500-я ошибка для пользователя. Это вообще неприемлемо.

Это важный аспект — с помощью очереди сообщений можно позволить сломаться какому-то куску вашей системы. Например, в одном из проектов, который мы разрабатывали, запись в базу шла через очередь сообщений. Это было очень удобно. Можно было на решардинг и на другие обслуживающие операции выключить один из кусков базы данных на какое-то время. Все это время у нас тупо росла очередь сообщений и что-то не записывалось. Потом, когда базу чинят и снова подключают, очередь рассасывается.

Таким образом, очередь сообщений позволяет вам еще и функционально развязать куски всей вашей экосистемы и позволить кому-то сломаться без общей деградации на время.



На рисунке изображено два сервиса, которые полностью независимы друг от друга. Поломка одного не приводит к поломке другого. Допустим, сервису А нужно отправить что-нибудь в сервис Б. Он ставит задачу во внутреннюю очередь сервиса А. Раздающий демон (своего рода выходные ворота сервиса А) разбирает внутреннюю очередь и рассыпает запросы вовне.

Входные ворота сервиса Б принимают запрос и пишут его во внутреннюю очередь сервиса Б. Воркеры сервиса Б обрабатывают задачи из внутренней очереди.

Подобная система не только практически неубиваема, она еще и восстанавливается после сбоя без потери данных :). Может сложиться впечатление, будто это нечто запредельное, но это не так. В крупных банковских системах подобные архитектуры встречаются на каждом шагу. Да и в веб-системах можно использовать что-то похожее для достижения независимости сервисов между собой.

В качестве резюме

Итак, мы изучили один из самых мощных паттернов в проектировании веб-систем — использование очередей.

Под очередями сообщений в проектировании веб-проектов могут пониматься две разные вещи. Во-первых, способ отложить задачу «на потом». Нам надо сделать что-то, но пользователю надо ответить прямо сейчас. Мы выходим за рамки PHP'шного «запрос — ответ» и делаем что-то чуть позже, чем отдали ответ.

Во-вторых, речь может идти о так называемой общейшине данных. У нас возникает межсервисная коммуникация, которая помогает разнести вызовы между сервисами, сделать их разными по времени и унифицировать общение между разными сервисами.

Удачи! В следующем номере самое сложное — масштабирование баз данных.

Урок 5.

Базы данных.

Последний пункт обязательной программы

Если твой сайт — это не домашняя страничка, то тебе нужно где-то хранить данные. Рано или поздно выясняется, что твоя СУБД перестает с этим справляться. Какие существуют подходы к масштабированию базы данных?

Подходов примерно столько же, сколько и для масштабирования фронтендов и бекендов, но ключевая мысль, с которой мы начнем — одна. Ты должен провести исследование предметной области, исследование потоков данных (подробно мы говорили об этом в прошлом уроке), и на основе результата этого исследования уже принимать решения о том, какие из видов масштабирования баз данных тебе подходят, какие нет.

Общего решения здесь нет. Подходит тебе синхронная репликация или нет? Подходит master-master или нет? Можешь ты поставить много баз данных и разбить данные между большим количеством экземпляров или нет? Все это зависит от конкретного приложения, от вашего пользователя и того, как ты хочешь показывать ему данные.

Чтобы говорить о конкретных решениях, нужно научиться анализировать предметную область своих данных. Для базы данных нужно определить модель представления данных, язык доступа к данным, на котором программист будет с ней общаться. Этую работу важно проделать на самом раннем этапе, хотя бы потому, что это напрямую влияет на выбор СУБД.

Различные типы баз данных

Для того, чтобы лучше понимать, как нам масштабировать базу данных, вспомним, какие, собственно СУБД у нас бывают? Если использовать классификацию по используемой модели представления данных, то получится четыре группы, представленных на слайде.

Типы баз данных

- **Реляционная модель:** данные в базе данных представляют собой набор отношений;
- **Иерархическая модель:** база данных состоит из объектов с указанием отношений родитель ⇔ ребенок;
- **Сетевая модель:** база данных со структурой в виде графа;
- **Объектно-ориентированная модель:** база данных, в которой данные представлены в виде моделей объектов.

Какие сейчас в этом смысле направления, тенденции? В принципе, мы немного откатились назад, лет на 30. Мы сейчас заново проходим графовые базы данных, сетевые базы данных, иерархические модели.

Итак, теоретически, выбор СУБД выглядит так: изучаете предметную область и определяете наиболее подходящую модель представления своих данных. На основе этого выбираете оптимальную для себя систему.



Это правильно, но на деле, в реальной разработке (и это правило относится ко всем сложным проектам) предпочтение нужно отдавать тем инструментам, которые знают ваши главные специалисты. Тем не менее, это не отменяет того, что специализированные решения можно применять для отдельных типов хранимых данных — в зависимости от их характера.

В данный момент для веба есть базы данных общего назначения. Это MySQL и PostgreSQL. Если рассматривать еще и специализированные решения, то список получится на 30-40 позиций. Это и Mongo, и Redis, и тот же Neo4j. Однако в общем случае для основных ваших данных вам нужен только MySQL или PostgreSQL.

Почему? База данных — это не только то, что вы видите. Это еще экосистема вокруг этого продукта, которая и заставляет его работать, расти и развиваться. Поясним на примере, почему это важно.

Допустим, вам хочется сделать полностью автоматический шардинг. Вы смотрите на автошардинг, сделанный в MongoDB, вам он нравится. Какие с этим могут возникнуть проблемы? Точно такие же проблемы, какие могут возникнуть с любой базой. У вас растет нагрузка, растет количество данных. MongoDB начинает, грубо говоря, тупить. И вот тут возникает главный вопрос — как и где придется решать такие проблемы? Это необходимо учитывать еще на этапе выбора СУБД.

Решение таких проблем упирается в развитость экосистемы вокруг используемого продукта. Есть ли сообщество, есть ли развитие продукта, есть ли кто-нибудь, кому я могу послать сообщение об ошибке или я использую СУБД на свой страх и риск? Именно поэтому лучше пользоваться более популярными продуктами с хорошей поддержкой.

Тюнинг запросов

Тюнинг запросов и оптимизация базы данных вообще — отдельная большая область знаний. Кратко перечислим основные направления, в которых можно проводить исследование.

Первая область связана с особенностями конкретного сервера базы данных, с его архитектурой. Сюда входят те буферы, кэши, которые использует сервер; механизм открытия/закрытия таблиц; различные особенности и так далее. Как правило, все эти параметры настраиваются.

Почему этим нужно заниматься? Приведем пример — настройки по умолчанию для СУБД PostgreSQL рассчитаны на работу всего-лишь с несколькими мегабайтами памяти — они крайне неэффективны. Отсюда следует, что настраивать базы данных необходимо.

Второе направление — особенности интерпретации и оптимизации SQL-запросов, которые применяются в данном SQL-сервере. Изучив эту сторону вопроса, можно значительно оптимизировать запросы программного обеспечения к базе данных. Нередко скорость обработки многократно увеличивается от введения одного небольшого индекса.

Также стоит обратить внимание на особенности операций с базой данных в общем. Чем отличаются операции выборки от операции вставки и какие конкретно физические действия придется совершать серверу базы данных при выполнении тех или иных запросов. Это — отдельная большая тема для разговора.

Третья плоскость, в которой стоит искать способы ускорить работу базы данных — структура базы данных, структура конкретных таблиц, индексирование и другие подобные вопросы.

Мы же поговорим только об одном — о том, какие запросы можно использовать в высоконагруженной системе, а какие нет?

Мы должны использовать все те же подходы, о которых говорили на предыдущих уроках — *share nothing* и *stateless*. Подход очень простой — представь сразу, что твои таблицы расположены не на одном, а на десяти серверах. Что они разрезаны, самые старые новости лежат на одном сервере, а новые на другом. Далее мы будем подробно говорить о механизмах подобного разделения. Но сейчас надо представить, что такое разделение уже произошло.

Теперь ответь на такой вопрос — как вы будете выполнять *join*'ы из таблиц, расположенных на разных физических серверах? Правильный ответ — вручную. Работа с базой данных в высоконагруженном проекте предполагает простые легкие конечные запросы. С заданием границ выборки, максимальным количеством извлекаемых элементов. Минимальное количество индексов, только самое необходимое, ведь индексы ускоряют выборки, но замедляют обновления БД.

Не рекомендуется использовать множество приятных внутренних механизмов СУБД. Объединения, пересечения — все это нужно делать в памяти бекенда. Нужно сделать *join* для двух таблиц? Сделай два отдельных запроса и перемножь результат в памяти. Не используй хранимые процедуры — как они будут работать, если исходные данные для них окажутся на разных серверах?

Следование этим простым рекомендациям может быть и замедлит несколько обработку страниц, зато сделает ее возможной, когда Ваш сайт вырастет.

Шардинг

Итак, ты смоделировал предметную область. Так или иначе, основная техника, которая используется при масштабировании СУБД — это шардинг.

Шардинг

Базовый принцип: те данные, которые в дальнейшем потребуются вместе, так же должны храниться вместе.

Примеры:

1. Пользователи;
2. Посты в сообществах;
3. Блоги;

Принципы разбиения данных на шарды:

1. Центральный диспетчер, знающий, что где лежит;
2. Хэш-функция, по ключу вычисляющая шард;
3. Хэш-функция, по ключу вычисляющая виртуальный шард + таблица соответствий виртуальных шардов реальным.

Основной принцип простой. У вас есть 100 миллионов пользователей в вашей социальной сети, чате и так далее. Всю информацию, относящуюся к первым десяти миллионам, храните на одной машине, ко вторым десяти миллионам — на второй машине. У вас 10 машин. Шардинг — это разбиение, нарезка ваших данных по машинам.

Ключевым вопросом тут является принцип шардинга. По какому критерию разбивать данные? По пользователям, по комментариям, по товарам, по каталогу, по категориям товаров? Как выбрать принцип разбиения — это отдельный большой вопрос. Нужно анализировать приложение и его бизнес-логику. Главный принцип — данные должны быть максимально связаны в одном шарде и минимально связаны между шардами.

Шардинг — это некий компромисс между масштабированием и удобством доступа, удобством аналитики.

При любом шардинге так или иначе таких правил, которое мы привели выше, (первые 10, вторые 10, третьи 10), множество. Это называется принципом построения шардинга. Когда строится такое разбиение, нужно учесть множество факторов, но главное — это твои данные.

Например, у тебя такая структура, что данные только добавляются, никогда не удаляются. Наиболее простым способом для этого будет некая парадигма с ящиками. Один шард — это ящик. Ящик наполнился — открыли следующий, ящик наполнился — открыли следующий. Таким образом, мы данные каждый раз добавляем в новую и новую машину. Потом мы уже думаем, что делать со старыми ящиками, которые, может быть, даже никогда не используются. Опустошать эти ящики, выкидывать, кидать в конец.

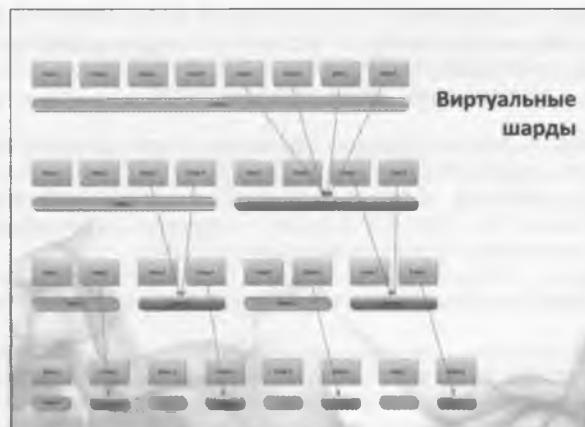
Рассмотрим второй принцип, который нужен в ситуации, когда характер нагрузки совсем другой. Допустим, что данные в каждом шарде могут расти или наоборот не расти по разным законам. Классический пример с Марком Цукербергом и Lady Gaga на Facebook. Если вы храните всё о Lady Gaga на компьютере № 69, рано или поздно этот компьютер переполнится.

Нужно думать, что делать со всеми этими данными. Или если вместе с Lady Gaga на этом же компьютере хранится 10 тысяч невинных обычных домохозяек, то рано или поздно хранение Lady Gaga на этом шарде приведет к тому, что домохозяйки получат низкое качество сервисов, потому что постоянно большой профиль нагрузки будет у Lady Gaga. Главная особенность такого сюжета — его непредсказуемость, поэтому нужна достаточно гибкая техника — виртуальные шарды.

Виртуальные шарды

Нужно предразбить пространство данных на заведомо огромное, но при этом равномерное по своей наполненности количество виртуальных шардов. Скажем, 100 тысяч виртуальных шардов. У тебя есть эта цифра, и изначально ты все эти шарды хранишь на небольшом количестве машин. Например, ты на каждой машине запускаете 10 MySQL'ей. В каждом MySQL'е ты запускаешь 100 баз данных, а всего у тебя 10, 20, 100 машин. Всё, предразбиение выполнено.

Постепенно вся эта система начинает наполняться, и можно достаточно беспроблемно (с помощью репликации) разнести данные на отдельные машины, на отдельные базы данных, на отдельные экземпляры серверов и так далее.



Эта техника называется “виртуальными шардами”. Разбиение данных по шардам — это некая договоренность, как мы будем класть элемент данных и как мы будем его потом искать. Универсального решения нет. Это некая договоренность между back-end’ом (бизнес-логикой) и системой хранения.

Виртуальные шарды — это некая прослойка, которая позволяет мне как back-end’у общаться всегда с конкретным шардом, не задумываясь о том, где физически находится этот виртуальный шард.

Получается, двойной процесс — принцип, похожий на схему работы виртуальной памяти в компьютере.. Пользователь вычисляется виртуально (например, по пользователю, по какому-то куску данных), определяется виртуальный шард. Затем берется некая таблица соответствий, по которой выясняется, где физически находится искомый шард.

Все это делается для того, чтобы в будущем, когда происходит рост каждого отдельно взятого шарда, мы могли легко и просто, не затрагивая ни бизнес-логику, ни программную часть, физически мигрировать данные с одной машины на другую.

При шардинге, как и при любой технике децентрализации все равно должна остаться какая-то центральная сущность, в которой хранится информация о том, как была проведена децентрализация. В нашем случае нужна информация, какой виртуальный шард на какой физической машине находится и какой пользователь к какому виртуальному шарду относится.

Выбор варианта центрального компонента зависит от сценария роста, от конкретного приложений. Один из них — иметь некоего диспетчера шардов, в котором хранится эта информация.

Второй — просто хранить в конфигурационном файле, если эта информация редко меняется. Вы просто распространяете этот конфигурационный файл по всему дата-центру, и везде, на любом конкретном компьютере у вас есть данные о том, что где лежит.

Третий способ — использование функционального принципа. У вас есть функция, которая однозначно выдает вам ответ. Все, что вам нужно, — это хеш-функция или некая комбинация хеш-функции и таблиц. Но принцип в том, что это функция. Это некие минимальные данные, которые редко (практически никогда) не обновляются. Вы можете использовать эти знания везде.

Центральный диспетчер

Есть компания “Badoo” (140 миллионов регистраций), сервис знакомств. По сведениям последнего года, они используют центральный диспетчер. У них нет никакой функции, которая по пользователю вычисляет, где конкретно хранится шард. В чем плюсы и минусы такого подхода?

Центральный диспетчер, при более сложной реализации, дает значительно лучшую утилизацию железа и возможность очень быстро обновлять серверный парк. Стоит центральный диспетчер и просто сообщает: «Вот тебе еще 10 серверов, заливай пользователей на них». Ты опять же можешь контролировать загрузку и знать, что это у тебя старая слабая машинка, на нее миллион пользователей, и баста. Вот эта новая, супер — на нее 10 миллионов пользователей.

Репликация

Каждый из серверов баз данных может выйти из строя. Надо быть к этому готовым, и тут на помощь приходит техника репликации.

Репликация — это средство связи между машинами, между серверами баз данных. По большому счету, это транспорт. С помощью репликации можно эти данные перенести с одной машины на другую либо продублировать на двух машинах. Это средство организации этого разбиения.

На каждую машину можно посыпать любой запрос. Все машины имеют общую копию данных — синхронная репликация. Эти данные с любой машины попадают на любую через некоторую трубу и эта труба позволяет иметь все эти реплики синхронными.

Какие тут принципиальные издержки, помимо того, что вам нужно резать данные? Каждые данные присутствуют в системе в двух-трех экземплярах. Каждая конкретная машина хранит то, за что непосредственно она отвечает, плюс она является запасом, бэкапом для какой-то другой машины.



Основной принцип использования репликации, который чаще встречается, заключается (опять же, как неожиданно!) в использовании особенностей запросов к базе данных. Наиболее вероятный сценарий использования базы данных — редкие операции обновления и частые запросы на чтение. Организуется простая схема, когда операции обновления идут на центральную систему, а оттуда реплицируются (копируются) на несколько серверов, которые выполняют запрос на чтение.

Партиционирование

У нас есть несколько подходов, чтобы сделать так, чтобы с масштабировать базу данных. Первое — шардирование, когда мы бьем данные по кусочкам, раскладываем их по выбранному критерию на машинах.

Второе — это партиционирование. Тоже бьем данные, но немного по другому принципу. То же самое, что функциональное разбиение бекендов. Все, что относится к форуму, лежит в одном месте. То, что относится к еще чему-то, лежит в другом месте. То, что относится к форуму, лежит в одной базе данных. То, что относится к новостям, — в другой базе данных.

Партиционирование

Функциональное разделение базы данных.

Разные данные хранятся в разных таблицах.

или

Разные данные хранятся в разных СУБД.

или

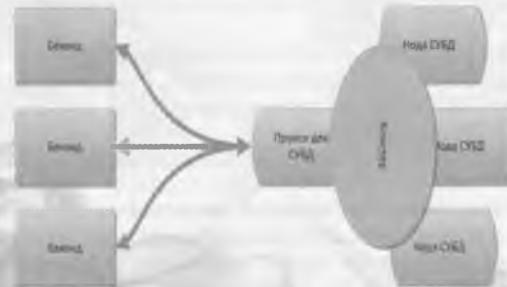
Разные данные хранятся в разных типах СУБД.

Потом начинаем двигаться еще дальше. Мы начинаем использовать особенности наших данных. Мы начинаем хранить, например, новости в реляционной базе данных, а что-то еще — в NoSQL'ной базе данных.

Кластеризация

Кластеризация, про которую упоминалось. Вы в каком-то смысле аутсорсите проблему масштабирования базы данных на разработчика этой самой базы данных или на разработчика кластера. Для вас кластер выглядит как единое целое.

Кластеризация



Часто кластер, действительно, выглядит как единое целое, но это скорее дань клиент-серверной архитектуре, дань тому, что изначально базы данных были большие и толстые.

Существует множество коммерческих и бесплатных кластерных решений. Ты покупаешь кластер, его настраивают и далее это решение самостоятельно. Все внутренние процессы могут быть тебе даже не известны. Это хорошо, с одной стороны — за тебя все настроили профессионалы. С другой стороны это плохо — у тебя нет возможности что-либо исправить в случае ошибки. Ты просто не знаешь, как эта штука работает.

Все тоже самое можно реализовать с помощью репликационной модели, когда вы просто-напросто соединяете базы данных в некую структуру. Данные в них движутся, существуют копии. Конкретные процессы репликации и шардинга в данном случае видны — поэтому возможно отстреливать лишние, при необходимости.

Денормализация

Рассмотрим еще один инструмент — денормализацию. Иногда для повышения эффективности хранения приходится размещать данные не самым оптимальным образом, то есть денормализуете. Например, можно их дублировать, можно хранить их в разных форматах. В примере с очередью модерации, как вы помните, мы хранили их в основной базе данных и отправляли еще куда-то. Система хранения, схема хранения, инструменты хранения отражают характер данных и предполагаемую модель их использования. Это происходит именно для того, чтобы ускорить обработку, ускорить построение страницы.

Денормализация данных

Денормализация — намеренное приведение структуры базы данных в состояние, не соответствующее критериям нормализации, обычно проводимое с целью ускорения операций чтения из базы за счет добавления избыточных данных.

Хороший пример того, как данные не денормализуются, а надо было бы, — это френд-лента в «Живом Журнале». Отсюда (во всяком случае, несколько лет назад) все их проблемы: низкая скорость работы и ограничение на количество френдов у одного пользователя. Дело в том, что каждый раз френд-лента строится нормальным, честным SQL-запросом: «Дай мне все сообщения всех моих друзей, отсортируй».

В Facebook это не так. Там каждое сообщение может храниться в нескольких миллионах экземплярах — именно для того, чтобы обработка данных происходила быстрее, чтобы быстрее показать пользователю news feed.

Это и есть денормализация данных. Ничего страшного, что данные хранятся в двух-трех экземплярах. В этом есть, естественно, оборотная сторона: нужно знать об этом, уничтожать или каким-то образом обрабатывать правильно. Но это позволит ускорить работу по построению страниц.

Чтобы осуществить денормализацию данных, придется немного поломать существующую модель представления данных, а это усложнит ее анализ. Для этого есть огромное количество решений, которые хранят данные в денормализованном виде, но могут представлять их в реляционном виде.

Особенности хранимых процедур в MySQL

Поговорим о достоинствах и недостатках хранимых процедур в MySQL. Напомним, что использовать этот инструмент в масштабируемой базе данных надо очень аккуратно.

Хранимая процедура — это текст, записанный в системную таблицу, который будет регулярно читаться из таблицы, компилироваться и кешироваться к скомпилированном виде в каждом соединении к серверу. Поскольку на каждый скомпилированный объект требуется от 80КБ оперативной памяти, при использовании большого количества хранимых процедур в большом количестве соединений к серверу надо рассчитывать на рост оперативной памяти, необходимой для MySQL. К примеру, при 1000 активных соединений, каждое соединение использует 20 хранимых процедур по 100КБ, необходимо до 2ГБ дополнительно оперативной памяти для хранимых процедур.

Несмотря на этот недостаток, использование хранимых процедур является распространённой практикой при доступе к данным в крупных веб-проектах по следующим причинам:

- дополнительный уровень внутренней безопасности. Прикладной программист, разрабатывающий сервис сети, вызывает хранимую процедуру, а не SQL запрос, и таким образом не имеет прав и может не знать непосредственной схемы данных
- возможности изменения схемы данных без изменения приложений. Меняется только уровень хранимых процедур.

Производительность процедуры напрямую зависит от её сложности. Распространённой практикой является создание хранимых процедур, инкапсулирующих не более 1-2х запросов к БД. Использование более сложных процедур не распространено, т.к.:

- пропорционально увеличивается задержка на выполнение процедуры
- отладка в случае неполадок усложняется, т.к. в MySQL нет интегрированного отладчика хранимых процедур, не говоря уже о том, что в production может быть ещё важно понять какой конкретно запрос в хранимой процедуре начинает выполняться медленно, и наличие большой процедуры добавляет сложности в поиске проблемы.

Важно учитывать то, что при изменении хранимой процедуры одновременно инвалидируются все кеши всех активных соединений. Это может привести к серьёзным «провалам» в производительности, т.к. все соединения одновременно будут пытаться пересоздать свои скомпилированные копии хранимых процедур. Это следует учитывать, и не планировать массированные изменения во время прайм-тайм нагрузок.

При репликации хранимых процедур MySQL использует так называемый «unrolling», т.е. в replication log попадает не непосредственно вызов хранимой процедуры (CALL GetUserComet(480145)) а запросы, выполненные внутри хранимой процедуры. Т.е. реплика не выполняет саму процедуру, а только те запросы, которые хранимая процедура использует и которые изменяют данные.

Необходимо также иметь ввиду, что алгоритм выполнения хранимых функций и триггеров в MySQL существенно отличается от описанного выше, т.е. не следует эти знания применять для хранимых функций и триггеров.

Последний пункт обязательной программы

Вот, наверное, и все основы масштабирования баз данных. Используйте все приемы разумно, в той мере, в какой необходимо.

Почему базы данных — это “последний пункт”, спросишь ты? Все очень просто — в предыдущих пяти уроках мы рассмотрели основные архитектурные модули типичного высококо-

нагруженного проекта — фронтенд, бекенд, базу данных. Для каждого мы перечислили типичные подходы к масштабированию. Ты можешь уже приступить к созданию своего собственного высоконагруженного проекта

Но разработать проект — это еще не все. Проект надо поддерживать, эксплуатировать, организовать правильный хостинг, правильный мониторинг. Вот об этих, сервисных, но совсем немаловажных аспектах пойдет речь в следующих уроках. До встречи!

Highload-инструкторы

Олег Бунин

Известный специалист по Highload-проектам. Его компания «Лаборатория Олега Бунина» специализируется на консалтинге, разработке и тестировании высоконагруженных веб-проектов. Сейчас является организатором конференции HighLoad++ (www.highload.ru). Это конференция, посвященная высоким нагрузкам, которая ежегодно собирает лучших в мире специалистов по разработке крупных проектов. Благодаря этой конференции знаком со всеми ведущими специалистами мира высоконагруженных систем.

Константин Осипов

Специалист по базам данных, который долгое время работал в MySQL, где отвечал как раз за высоконагруженный сектор. Быстрота MySQL — в большой степени заслуга именно Кости Осипова. В свое время он занимался масштабируемостью MySQL 5.5. Сейчас отвечает в Mail.Ru за кластерную NoSQL базу данных Tarantool, которая обслуживает 500–600 тысяч запросов в секунду.

Максим Лапшин

Решения для организации видеотрансляции, которые существуют в мире на данный момент, можно пересчитать по пальцам. Макс разработал одно из них — Erlvideo (erlyvideo.org). Это серверное приложение, которое занимается потоковым видео. При создании подобных инструментов возникает целая куча сложнейших проблем со скоростью. У Максима также есть некоторый опыт, связанный с масштабированием средних сайтов (не таких крупных, как Mail.Ru). Под средними мы подразумеваем такие сайты, количество обращений к которым достигает около 60 миллионов в сутки.

Константин Машуков

Руководитель отдела анализа и синтеза компании «Лаборатория Олега Бунина». Константин пришел из мира суперкомпьютеров, где долгое время «пилил» различные научные приложения, связанные с числодробилками. В качестве бизнес-аналитика участвует во всех консалтинговых проектах компании, будь то социальные сети, крупные интернет-магазины или системы электронных платежей.

Секция «Архитектуры»

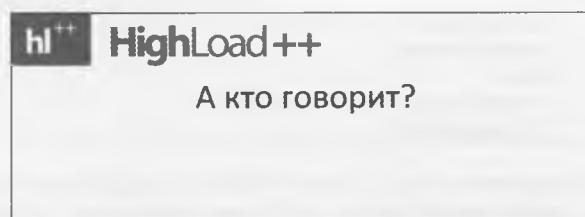
Впервые в рунете: сказ о 100М писем в день

Андрей Сас

Господа, рассаживайтесь. Время доклада подошло. Из уважения к докладчику и к аудитории постараитесь, пожалуйста, побыстрее.



Андрей Сас: Всем привет! У меня весьма интересная тема. Впервые в Рунете я буду рассказывать об отправке ста миллионов email-писем каждый день. Мой доклад основан на практике, которую я приобрел в компании «Badoo».



Для начала немного подробнее о том, кто я такой и почему меня стоит слушать.

HighLoad++

А кто говорит?

Я:

- руковожу развитием почтовой рассылки, могу знать не все детали;

Во-первых, я руковожу разработкой всей почтовой инфраструктуры и, если мне будут задавать вопросы про какие-то детали реализации, я могу не ответить, т.к. просто их не знаю.

HighLoad++

А кто говорит?

Я:

- руковожу развитием почтовой рассылки, могу знать не все детали;
- не админ (извините!).

Во-вторых, я не админ. Поэтому когда я буду говорить про то, как у нас конфигурируется софт или какое у нас железо, я буду рассказывать своими словами. Все будет понятно, но админы, наверное, меня невзлюбят.

HighLoad++

А кто говорит?

Я:

- руковожу развитием почтовой рассылки, могу знать не все детали;
- не админ (извините!).

Хвастаюсь:

- полгода без пролёжек почтовой инфраструктуры;

Теперь немного про то, почему меня стоит послушать. Во-первых, в нашей почтовой инфраструктуре не было никаких «пролежек» на протяжении полутора лет.

HighLoad++

А кто говорит?

Я:

- руковожу развитием почтовой рассылки, могу знать не все детали;
- не админ (извините!).

Хвастаюсь:

- полгода без пролёжек почтовой инфраструктуры;
- 97% доставки в инбокс;

Второе достижение, которым я горжусь, — наши письма попадают в папку «Входящие» в 97% случаев. Учитывая, что рассылаемся мы не по России (в основном, это почтовики «большой тройки» — Hotmail, Yahoo и Gmail), это достаточно приличный результат.

Наконец, я хотел сделать отдельный замечательный слайд с цифрой 42 на весь экран, но не получилось. Получилась цифра 25.

Наконец, я хотел сделать отдельный замечательный слайд с цифрой 42 на весь экран, но не получилось. Получилась цифра 25.

hl⁺⁺ HighLoad++

А кто говорит?

Я:

- руководжу развитием почтовой рассылки, могу знать не все детали;
- не админ (извините!).

Хвастаюсь:

- полгода без пролёжек почтовой инфраструктуры;
- 97% доставки в инбокс;
- среднее время доставки почты – 25 с.

Это среднее количество секунд, за которое пользователь получает от нас письмо.

hl⁺⁺ HighLoad++



В общем, я очень скромный.

hl⁺⁺ HighLoad++

О чём буду рассказывать?

Итак, я буду говорить о том, как сделать так, чтобы вы слали много писем, хорошо себя чувствовали и нормально спали по ночам.

При этом базовый вопрос: как шлют почту.

hl++

HighLoad++

О чём буду рассказывать?

А как люди отправляют почту?

Когда готовился к этому докладу, не хотел повторяться и начал просматривать, какие уже были выступления на эту тему на русскоязычных конференциях за последние 3-4 года. Я нашел ровно ноль докладов на эту тему.



Соответственно, я сначала подумал: «Наверное, там большая-большая страшная тайна, раз никто не говорит». У меня сейчас немного другое мнение. Мне кажется, в России никто не углубляется в данную проблематику. Стоит себе почтовый сервер. Команда «mail» в PHP выполнилась, что-то куда-то ушло — все круто. Либо же люди считают: «Что там рассказывать? Все просто».

hl++

HighLoad++

О чём буду рассказывать?

Как отправлять много-много писем,
быть уверенным в себе
и спать по ночам спокойно

Какие же задачи ставит передо мной бизнес?

hl⁺⁺

HighLoad++

Бизнес-задачи

1. Предоставить прозрачный API программистам.
2. Обеспечить отправку почты в объемах до 100М писем в день.
3. Обеспечить доставку почты в инбоксы в 95%+ случаев.

Во-первых, мой отдел должен предоставить прозрачный API нашим программистам, которые делают новые типы писем. Другими словами, задача программиста сводится к тому, чтобы подготовить HTML и текстовую версии письма, указать заголовок и адресата, а про все остальные заботы ему знать не нужно.

В частности, программиста, разрабатывающего фичу, не должно волновать, как десятки миллионов писем каждый день будут перебираться с машин, где они создаются, на почтовые сервера, как мы решаем проблемы с доставкой в отдельные почтовики, каким образом будет собираться статистика и вестись мониторинг.

Вторая задача заключается в том, чтобы обеспечить возможность слать до ста миллионов писем каждый день.

В-третьих, нам нужно, чтобы абсолютное большинство писем доставлялось в инбоксы пользователей.

hl⁺⁺

HighLoad++

О чём НЕ буду рассказывать?

Как сделать так, чтобы ваши письма не попадали в Spam

Важный момент. Я говорю про техническую сторону: как справиться с ситуацией, когда у вас в день отправляется сто миллионов писем. Но не буду говорить и комментировать то, как мы избегаем попадания в Spam. Это наше конкурентное преимущество, про которое мы говорить не готовы.

hl⁺⁺

HighLoad++

А что в этом сложного?

Казалось бы:

- поднял MTA
- сделал mail()
- PROFIT!

Наверное, сейчас многие задаются вопросом: в чем, собственно говоря, сложность? У нас есть некий почтовый сервер (MTA), мы его подняли, сконфигурировали, сделали команду «mail», она сказала «true» — все, замечательно, проблема решена.

hl⁺⁺ **HighLoad++**

А что в этом сложного?

Казалось бы:

- поднял MTA
- сделал mail()

Однако:

- отправка 1 письма = обработка 1 динамического хита

Но, на мой взгляд, вы упускаете важные вещи.

Во-первых, чтобы отправить письмо, потребуется примерно столько же вычислительных ресурсов, сколько понадобилось бы, чтобы обработать динамический хит — например, показать главную страничку badoo.com.

hl⁺⁺ **HighLoad++**

А что в этом сложного?

Казалось бы:

- поднял MTA
- сделал mail()

Однако:

- отправка 1 письма = обработка 1 динамического хита
- а ведь письма ещё нужно генерить

Во-вторых, это письмо нужно еще и сгенерировать. Эта задача требует ресурсов того же порядка, что и отправка.

hl⁺⁺ **HighLoad++**

А что в этом сложного?

Казалось бы:

- поднял MTA
- сделал mail()

Однако:

- отправка 1 письма = обработка 1 динамического хита
- а ведь письма ещё нужно генерить
- смелость пойти и узнать правду!!1111

Наконец, нужно быть достаточно смелым, чтобы пойти и посмотреть, что у вас происходит с этими письмами.

Если вы зададитесь этим вопросом, то обнаружите, что каждое сотое письмо у васбитое, в каждом десятом ссылки указывают не туда или что-то другое иногда срабатывает неправильно. Если email является для вас важным каналом привлечения пользователей, то такие ошибки недопустимы.

hl⁺⁺

HighLoad++

Откуда взялась цифра 100M?

- 50M – каждый день
- 70M – в пике
- 100M – «пасаны ваше ребята. молодцы, могёте!»
- 150M – с новой поставкой серверов

Сразу объясню, откуда взялась цифра 100 миллионов, обозначенная в названии доклада.

Я ей очень горжусь. Ни у кого больше в заголовке доклада нет большей цифры на этом «HighLoad».

Во-первых, в любой день мы отправляем не менее 50 миллионов писем. У нас были дни, когда мы проводили специальные кампании, и получалось 70 миллионов. Зная, какая у нас сейчас архитектура с точки зрения железа, я уверен, что при ста миллионах в день все будет хорошо. Скоро, в согласии с моими планами по построению наших почтовых кластеров, мы сможем слать до 150 миллионов в день.

hl⁺⁺

HighLoad++

Особенности больших проектов

Наши мантры:

- нужно отправлять письма асинхронно

Почему в больших проектах почта — это сложно? Какие там есть особенности? Хочу сразу отметить, что когда готовил доклад, меня старшие товарищи напутствовали: «Не надо делать доклад, который будет bull shit'ом, как часто бывает». Послушал что-то: круто, классно, будем делать. Приходишь на работу — начинаешь пытаться придумать, как это вообще может работать, и ничего непонятно! Все повисло в воздухе.

Я постараюсь говорить о практике, как это сделано в «Badoo». В максимальном количестве мест говорить о конкретной реализации.

Первый момент, который возникает в больших проектах. Нужно делать асинхронно. Все знают «highload», «асинхронность», еще пару очень важных слов. Все очень круто. В случае с отправкой почты все вроде бы просто понимают, что не нужно команду «mail» делать там (в том же скрипте), где возникла необходимость отправки письма.

hl⁺⁺

HighLoad++

Особенности больших проектов

Наши мантры:

- нужно отправлять письма асинхронно
- по-настоящему (вдвойне) асинхронно!

Но по опыту «Badoo» мне кажется, круто, когда у вас асинхронность двойная. Мало того, что вы не отправляете письмо в том месте, где понадобилось его отправить, вы еще и отделяете генерацию письма от ее отправки.

Подробнее чуть-чуть попозже расскажу.

hi⁺⁺ **HighLoad++**

Особенности больших проектов

Наши мантры:

- нужно отправлять письма асинхронно
- по-настоящему (вдвойне) асинхронно!
- требуется балансировка между серверами

Когда у вас маленький проект — это, скорее всего, один почтовый сервер. Вы забили его IP'шку в какой-то конфиг вашего локального Postfix'а — и все замечательно.

Когда у вас большой проект — много почтовых серверов. Значит, нужно между ними как-то балансировать нагрузку, учитывая тот факт, что они разные по производительности, по эффективности и так далее.

hi⁺⁺ **HighLoad++**

Порядок отправки письма

1. Появляется необходимость создать письмо.
2. Постановка в очередь на создание письма.
3. Генерация письма по задачам из очереди на создание.
4. Постановка в очередь на отправку.
5. Отсыпка письма из очереди на отправку.

Давайте теперь поподробнее по каждому пункту пройдемся.

Про порядок отправки письма.

В нашем представлении в «Badoo» все выглядит следующим образом.

Пункт номер раз. Предположим, один пользователь отправил другому сообщение. Нужно послать последнему уведомление в почту, письмо: «У вас сообщение пришло».

Соответственно, в этот момент у нас просто откладывается какая-то заявка в очередь на создание уведомления о новых сообщениях. Она там тихо-спокойно себе живет.

Есть отдельно существующий скрипт, который в такого рода очереди (каждый свой, соответственно) генерит письма про новые сообщения. Он забирает из этой очереди заявку, генерит письмо и сам ничего не отсылает, команду «mail» не делает.

Вместо этого он откладывает уже сгенеренное письмо в некую очередь на отправку. Если, предположим, у вас тупит MTA (долго отвечает на ваши запросы), у очереди не возникнет тупняка в скрипте, который генерирует письма. Он возникнет в другом месте, что для вас хорошо.

Есть очередь на отправку. Ее разгребает отдельный скрипт, который для каждого из генеренных писем, по сути, только команду «mail» делает и куда-то отправляет.

hl⁺⁺ HighLoad++

Очередь на отправку

Наша реализация – на файлах. Преимущества:

1. Возможна работа без внешних сервисов.
2. Простота манипулирования письмами.
3. Легко получить статистику / логи.
4. Просто реализуются многократные попытки отправки.

Теперь подробнее про эту очередь на отправку, которую я только что упомянул. Реализована она очень просто – на файлах. Нам эта реализация нравится по ряду причин (конечно, кроме того, что она исторически сложилась).

Первый момент. Для отправки таких писем вам не нужно никаких внешних сервисов. Вы не завязаны ни на MySQL, ни на Mongo – ничего. Есть живой MTA, есть сервер, на котором у вас лежат письма, и сеть между ними работает замечательно. Проблема решена, вы можете все отправить. Пусть все остальное лежит – неважно. Письма у вас уходят.

Второй момент. Очень просто манипулировать письмами, когда они в виде файликов. Любой программист с ходу знает, как с ними работать. Там есть локи, вы можете переименовывать, перемещать, копировать – все что вам угодно. Все очень просто.

Наконец, со статистикой и логами тоже никаких проблем не возникает. У вас есть какое-то проблемное письмо. Вы просто скопировали его в какую-то папочку для проблемных писем, и потом с ними разбираетесь. Все буквально «на пальцах».

Точно так же операции многократных отправок.

Не удалась первая отправка – скопировали файлик в папочку, где вторая, третья и так далее попытки будут происходить, из которой эти файлик будут браться.

Все очень просто. Надеюсь, никого не удивил.

hl⁺⁺ HighLoad++

SSMTP вместо sendmail

Это SMTP-клиент, эмулирующий работу sendmail.
Нам он нравится, т.к.:

- ничего лишнего, только отсылает письмо в MTA (hub)
- супер простой конфиг
- мы его слегка допилили (таймауты + параметры)

Важный момент.

Мы не используем полноценный MTA на тех машинах, которые генерят почту, потому что смысла в этом не видим. Вместо этого мы используем утилиту SSMTP, которая умеет делать только одно. Грамотно пересыпал почту в настоящий MTA, в настоящий почтовый сервер, который потом уже будет заниматься доставкой. Он умеет делать только это, нам нравится.

Благодаря такому маленькому функционалу у него очень простой конфиг. Разберется и не админ, человек, который в первый раз видит.

Наконец, мы его слегка допилили, таким образом решив проблему, что в нем не было таймаутов, что он не умел через командную строку принимать какие-то параметры. Теперь он нам очень-очень нравится. Я действительно уверен, что никакого смысла на локальные машины, генерящие письма, ставить полноценный MTA, какой-либо Postfix, sendmail смысла нет.

The screenshot shows a slide from a presentation. At the top left is the logo 'hl++' followed by 'HighLoad++'. The main title is 'Балансировка между МТА'. Below the title, there are two sections of bullet points:

- Первая версия – на базе железки F5 LTM:**
 - weighted round robin
 - SMTP мониторинг
- Текущая реализация – скрипты на PHP + мониторинг от F5 LTM:**
 - автоматическое управление всей балансировкой
 - красивый веб-интерфейс
 - скрипач (админ) не нужен!

Я говорил, что появляется балансировка в больших проектах. Расскажу про две версии, которые у нас были в «Badoo», как мы это делали.

Версия номер раз. Она в начале этого года была заменена на более совершенную. Она была сделана на базе железки, такой специальный сервер от компании F5, называется LTM (Local Traffic Manager).

Что он умеет делать. Есть какая-то виртуальная IP'шка на LTM, к которой вы делаете коннект, желая отправить почту. При этом эта железка LTM знает, что к этой IP'шке относится пул из этого, этого и вот этого сервера.

Что делает LTM. Он прозрачно проксирует ваш запрос к конечной машине, конечному, реальному MTA. LTM умеет делать это с учетом весов по алгоритму round robin. Что важно: он мониторит в целом то, насколько жив сервер именно с точки зрения SMTP протокола. Если сервер не отзывается как надо, он его временно вычеркивает из этого списка и не подает на него трафик, пока он снова не предъявит признаки того, что он жив.

Схема вроде бы хорошая, интересная, но мы ее немного допилили, сделали так, что теперь эта логика по тому, куда слать, у нас лежит в PHP. При этом мы сохранили мониторинг от железки LTM. Почему нам этого так захотелось. Потому что теперь мы имеем совершенно полный контроль над тем, что нам делать в каком случае, когда мы хотим почту слать с этой машины, когда не слать.

Например, такую характеристику, как размер очереди на отдельной машине LTM не сможет учесть.

Мы отследили это с помощью PHP, закрыли какую-то IP'шку и все. Трафик туда не идет – проблема решена. Такие мелочи (не нужен красивый интерфейс, мы можем любые уведомления по почте, по sms привязать и так далее) – все в плюс.

Главное – «не нужен скрипач» (админ), который может потратить час, два, три (а в выходной день, может быть, больше), чтобы выкинуть какую-то машину из этого списка, в котором мы больше не хотим видеть машину.



Автоматизация при отправке

Хорошее место, чтобы делать добрые дела:

- подстановку мониторинговых параметров в ссылки
- подстановку картинок для мониторинга открытых
- проверку целостности и корректности письма
- подстановку технических заголовков
- и даже проверку работоспособности ссылок!

Хочу осветить важный момент.

Есть такая возможность — много чего автоматизировать при отправке. У вас, наверное, есть какой-то участок кода, который выполняется для всех отправляемых писем. По-моему, это очень крутое место для того, чтобы сделать кучу вещей, которые иначе пришлось бы делать с большими трудозатратами для каждого нового письма в отдельности.

Просто на нашем примере, что мы там делаем.

- Мы умеем во все ссылки подставлять какие-то мониторящие параметры, чтобы понимать, из какого письма был сделан клик.
- Мы умеем автоматом подставлять картинки для мониторинга того, как наши письма открывают.
- Можем проверять корректность писем, целостность, наличие ссылки на отправку.
- В конце концов, мы можем даже эти ссылки автоматом туда подставлять.
- Куча технических заголовков, которые вам могут понадобиться. Тоже можно воткнуть.

Просто на заметку, что там можно делать много всяких хороших вещей.



Железо почтового кластера

Кластер — это по 10 машин на каждой из площадок:

- 8 отправляющих
- 2 принимающих

Важна только дисковая подсистема:

- in memory решения не пробуем, не хотим терять письма
- штатно 4 x SAS 10k в RAID 1+0, плюс контроллер с «батарейкой»
- для тестов 4 x SSD в RAID 5

Немного расскажу про железо, которое у нас есть. Тут разом высветились все мои секреты. У нас две площадки: одна в Европе, другая — в Северной Америке. На каждой имеется почтовый кластер. 10 машин на одной, на другой — чуть побольше. Скоро будет одинаковое количество, и вообще машин станет больше. Но соотношение отправляющих письма машин к принимающим и обрабатывающим машинам приблизительно восемь к двум.

Теперь про то, какие машины по нашему опыту стоит брать, использовать и так далее. Если мы посмотрим на CPU usage на этих машинах, то обнаружим, что он практически не используется, даже при том, что мы все наши письма подписываем DKIM'ом и DomainKeys'ом. Там CPU usage получается на каком-нибудь процессоре типа слабенького Core 2 Duo двухъядерный, наверное, 2-3% в пике. В общем, процессор не важен.

То же самое могу сказать про память. Два гигабайта (ну, четыре в пике), которые могут потребляться реально MTA и системой в момент самой высокой загрузки.

Важным остается один параметр, в который вы можете уткнуться, пытаясь отправить много-много почты. Это ваша дисковая подсистема. Мы готовим ее следующим образом. На всех серверах у нас стоит собранный из четырех SAS-дисков на 10 тысяч оборотов каждый, RAID 1+0. Он дает уверенность в том, что мы не потеряем письма. Это одна из причин, почему не пробуем in-memory решения. Также дает более высокую скорость работы.

Важно сказать (наверное, для админов это очевидно, а для остальных людей — не очень), что хороший RAID-контроллер может позволить вам значительно ускорить работу всей этой подсистемы. Есть такая штука, называется RAID-контроллер с батарейкой, который позволяет делать следующее. Вы записываете данные не собственно на диск, а складываете в некоторую плашку памяти, которая к этому RAID-контроллеру присоединена. В этот момент вам RAID-контроллер говорит: «Я все записал». Вы ничего не ждете, идете дальше, а в реальности потом с этой некоей памяти RAID-контроллер записывает информацию на диск.

«Причем здесь батарейка?» — спросите вы. Это такая простая защита от того, что нам вырубили питание, и инфа не успела дозаписаться. Она будет лежать какое-то продолжительное время на этой плашке, пока сервер не поднимут, и тогда RAID-контроллер дозапишет на диски данные.

Приехала одна машина на четырех SSD дисках для тестов, которую мы хотели собрать в RAID 5. Но ее вендор забыл, поэтому она не приехала, и я вам про нее не расскажу.

HighLoad++

Как тюнить MTA?

- оптимизировать файловую систему
- увеличить число SMTP воркеров
- увеличить число DNS воркеров
- поставить локальный кэшер DNS-запросов
- раскладывать очередь по большому числу директорий
- увеличить лимиты на число соединений к одному MX серверу
- выставить лимиты на число писем в сессии

Теперь общие моменты по тому, как мы тюним MTA, наши почтовые сервера, чтобы они рассыпали максимум возможного. Тут куча всяких пунктов. Самое главное, с чего стоит начать, — это оптимизировать файловую систему, которую вы используете. Это единственное, во что вы можете серьезно упереться. Здесь я бы порекомендовал в первую очередь посмотреть на флагочек noatime.

Теперь говорим про настройку MTA. Тут есть всякие разные параметры, которые от почтового сервера к почтовому серверу могут быть, отсутствовать. Это число SMTP-воркеров. Грубо говоря, сколько вообще тредов, процессов занимается отправкой почты в сторонние почтовики, и число DNS-воркеров. Такой термин есть в «Communigate». У него отдельные потоки для получения MX'ов. Во сколько потоков все это пытается обрабатываться. Если у вас много DNS запросов (а на почтовом сервере их много), то вам неплохо было бы поставить локальный кэш их результатов. Мы используем unbound.

Такой интересный момент. У вас иногда могут начаться проблемы с почтой. У вас, например, миллион писем накопится. Сеть наружу не работает или какой-нибудь Hotmail не хочет от вас писем принимать — у вас набрался миллион писем. Если не разложите их по большому числу папочек, то можете получить вполне известные тормоза по тому, как быстро будут читаться файлики в директории. Это чуть ли не основное действие, которое делает почтовый сервер.

Есть еще две вещи, которые завязаны, скорее, на принимающую сторону (Hotmail, Gmail, Yahoo). У каждого из почтовых сервисов есть свое ограничение на то, сколько одновременно коннектов вы можете к их MX'ам установить или сколько писем они готовы принять в рамках одной SMTP-сессии. Эти вещи нужно узнать, понять и проставить правильно для каждого из ваших важных почтовиков, куда вы шлете почту.

hl⁺⁺ HighLoad++

Наши MTA

Исторически – Communigate Pro:

- надёжный
- ОЧЕНЬ быстрый

Для «проблемных» почтовых сервисов – Postfix:

- более конфигурируемый
- есть возможность доработать напильником

Теперь от общих слов к конкретике. Мы в компании «Badoo» используем два почтовых сервера, два софта. Это «Communigate Pro», который у нас исторически. Мы не считаем, что от него нужно уходить, убегать, что он плохой. Он очень надежный и очень быстрый с точки зрения того, сколько писем в единицу времени он может отправить.

Наконец, у нас есть Postfix'овые сервера, которые мы используем для того, чтобы доставлять почту в какие-то проблемные почтовики. Просто потому что он более конфигурируемый, и при желании, если мы найдем какую-то страшную багу в нем, попросим С'шников. Они доработают его напильником, все будет хорошо.

hl⁺⁺ HighLoad++

Хм, Communigate Pro?..

Но ведь есть Postfix, Exim,
Hurricane, Message Systems, Zrinity
и даже Exchange!

Вы, наверное, услышав слова «Communigate Pro», немного удивились. Часть подумала: «Что это такое?». Часть: «Почему?». Есть же Postfix, Exim, есть какие-то дорогущие специализированные MTA — Hurricane Systems, Zrinity, какой-нибудь eCelerity, используемый «Facebook»ом. Наконец, можно было на Exchange попробовать все это сделать.

hl⁺⁺ HighLoad++

Корпоративный комбайн

- Email – первоначальный, основной продукт
- Calendaring
- VoIP
- IM
- File storage
- IP PBX
- Presence

Формально «Communigate» – это действительно такой корпоративный комбайн, который умеет делать Email, LDAP, VoIP. Он умеет Jabber сервер из себя делать, еще какие-то слова, которые я даже не понимаю, что значат.

Но важный момент здесь в том, что его первоначальным продуктом был именно MTA. Он был написан очень классно. Был написан русскими. Он показывает результаты.

The screenshot shows the HighLoad++ software interface. At the top, there's a logo with 'hi ++' and the text 'HighLoad ++'. Below it, a section titled 'А кто им пользуется?' (Who uses it?) displays logos of various companies: Deutsche Telekom, NETGEAR, SoftBank, at&t, BT, Comodo, SingTel, and Telkom Indonesia.

К сожалению, не в этих компаниях, которые висят на сайте «Communigate», кто нами пользуется. Зато показывает хорошие результаты у нас в «Badoo»'шечке.

The screenshot shows the HighLoad++ software interface. At the top, there's a logo with 'hi ++' and the text 'HighLoad ++'. Below it, a section titled 'Однако, цифры' (However, numbers) displays the following text: 'На старой машине с 1 диском SCSI 10k:' followed by a bulleted list: '• 5 миллионов писем в сутки' and '• до 100 писем в секунду в пике'.

Грубо говоря, у нас была старая-старая машина, на которой был один SCSI диск на 10 тысяч оборотов. Никакого RAID'a, ничего. Он умудрялся при этом посыпать 5 миллионов писем в сутки стабильно. В пике это было 100 отправленных успешно (Hotmail, Yahoo, Gmail) SMTP-сообщений, писем. По-моему, это очень классный результат. Если кто-то реально имеет дело с тем, сколько писем можно доставить, ему, я думаю, эта цифра весьма понравится.

The screenshot shows the HighLoad++ software interface. At the top, there's a logo with 'hi ++' and the text 'HighLoad ++'. Below it, a section titled 'Общее ощущение' (General feeling) lists pros and cons:

- + чрезвычайно стабилен, вплоть до LA = 200 и очереди в 1M писем
- + высокая производительность отправки писем
- + не требователен к памяти и CPU
- + достаточно настроек для большинства проектов
- * платный
- нет возможности менять настройки для разных почтовиков
- нет возможности допилить самим
- проблемы с выводом некоторых видов статистики

Общее ощущение от «Communigate». Во-первых, он чрезвычайно стабилен. Даже при каком-нибудь LA = 200 на четырех ядрах и очереди на один миллион писем он живет. Это практика того самого сервера, который я написал перед этим. Он живет и отсылает те самые 5 милли-

нов писем. LA = 200 меня лично впечатляет всегда.

Он, конечно, doch раз в пару дней, все бывает. Но в остальное время все работало, все замечательно, все классно. Ребята, которые знают, о чем говорю, смеются.

Высокая производительность, которой я уже успел похвастаться. Нетребовательность к памяти и CPU. Грубо говоря, я ни разу в жизни не видел, чтобы «Communigate» даже на самой нагруженной системе кушал больше, чем два гига памяти вместе с системой. На память вообще плевать.

Кроме того, настроек у него достаточно для того, чтобы решить проблему доставки в отдельные почтовые сервисы для абсолютного большинства проектов. Только если вы, как мы, готовы заморачиваться на то, что у вас 1% писем туда не ходят, и вы готовы ради этого поставить какой-то отдельный тип MTA (в нашем случае Postfix), ok, круто, вперед.

Я обозначил пункт «платный» не плюсом и не минусом по одной простой причине. У «Communigate» лицензия рассчитывается из числа юзеров, использующих веб-морду, поскольку это корпоративный продукт. Соответственно, если вы надумаете его использовать, лицензия вам самая минимальная подойдет. Сколько вы ни отсылайте с него писем — миллион в день, 5 миллионов, до 20-ти его разгоните — будет одна и та же маленькая стоимость лицензии. За рекламу они мне, надеюсь, что-нибудь пришлют.

Теперь про минусы системы. Во-первых, вы не можете настроить ее по-разному для каждого почтового сервиса. Соответственно, проблемные будут пользоваться теми же самыми правилами, что и супер популярные (Yahoo, Gmail, Hotmail).

Нет возможности его допилить самим. Просто исходников «нема». Наконец, есть совершенство дурацкие проблемы с тем, как он отображает статистику в своей веб-морде. У него есть веб-морда, вы не поверите. Он любит показывать первую тысячу пунктов без возможности сортировки. Все остальные 20-30 тысяч вы никак не можете увидеть. Вот просто так.

The screenshot shows a web-based monitoring interface for 'HighLoad++'. At the top, there's a dark header bar with the text 'hi ++' on the left and 'HighLoad ++' in large white letters. Below the header, the main title 'Статистика и мониторинг' (Statistics and Monitoring) is displayed in bold black text. Underneath the title, a message in Russian reads 'Пока не измеряешь – не контролируешь.' (Until you measure it, you don't control it.). A section titled 'Что было в начале?' (What was at the beginning?) contains a bulleted list of monitoring features:

- графики по числу писем в очереди на каждом почтовике (graphs by the number of messages in the queue on each mail server)
- сколько каких писем отправили за сутки (how many messages were sent in a day)
- статистика по LA / CPU usage / Memory usage в Zabbix (statistics for LA / CPU usage / Memory usage in Zabbix)

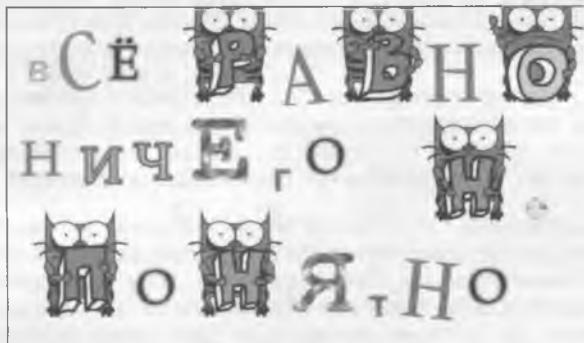
Все, что я перед этим рассказал, это про то, как у нас архитектурно все устроено (естественно, в общих деталях). Мое мнение такое. Даже если вы полностью скопируете эту архитектуру и будете даже слать не наши стабильные 50 миллионов писем в день, а 5, у вас возникнут проблемы. Проблемы эти будут. Пока вы не мониторите, что у вас происходит, вы не можете гарантировать, что вы вообще контролируете ситуацию.

Расскажу, что у нас было вначале, два года назад, когда я начал заниматься почтовыми вещами в «Badoo». У нас было три вида статистики. Во-первых, были графики, которые показывали, сколько писем стоит в каждой конкретной MTA машине. 200 тысяч в одной, 50 тысяч в другой и так далее.

Второй момент. У нас были графики по тому, сколько писем каждого типа (100 тысяч регистраций в день, 500 тысяч писем про то, что у нас новые сообщения и комменты и так далее). Сколько мы отправляем каждого типа за сутки.

Наконец, у нас был Zabbix мониторинг, который показывал потребление CPU, памяти, какие-то операции по диску, сколько их и так далее.

В результате, когда возникала проблема, на мой (и не только) взгляд, происходило вообще что-то странное. Мы смотрели на эти графики и ни фига не понимали.



В результате мы порешили и сделали несколько вещей, которые нам помогли и помогают до сих пор больше всего.

hl++ HighLoad++

Чего не хватало больше всего?

- число файлов, ожидающих отправки в MTA
- число ошибок отправки писем в MTA
- среднее время отправки письма в MTA
- самые загруженные отправкой почты скриптовые машины
- среднее время доставки почты в отдельный почтовый сервис

Во-первых, я упоминал, что очередь на отправку писем... Есть машина, которая генерит письма, и на ней сгенеренные письма складываются в файлики. В какой-то момент я понял, что я вообще понятия не имею, сколько всего в нашей системе писем, еще не отправленных в наш MTA-сервер, лежит на дисках. Такую статистику решили делать.

Второй момент. Надо было считать, сколько ошибок при отправке MTA у нас случается. Без этой информации тоже не очень понятно. Может, у нас там по 10 раз все перепосыпается.

Третий момент. Неплохо бы знать, сколько времени в среднем требуется на то, чтобы письмо сгенерили, положили, и сколько пройдет до того, как оно попадет в наш MTA, который будет его отправлять в Hotmail или еще куда-то.

Есть информация о том, как загружены наши сервера, которые генерят почту. Есть какой-то лимит, который с каждой машины может быть отправлен в MTA в единицу времени. Нужна была такая информация. Сделали.

Наконец, среднее время доставки. Я говорил, что хотел, чтобы получилось 42, но получилось круче — 25. Это оно. Сколько времени требуется (в секундах), чтобы доставить письмо Hotmail'овскому юзеру, в Mail.Ru и так далее. Мы эти цифры получили и теперь имеем представление, что происходит.



Как реализовали?

Число файлов, ожидающих отправки в MTA:

- просто считаем файлы! ☺

Число ошибок отправки в MTA:

- просто считаем файлы! ☺

Чтобы побольше практики, которую я обещал, расскажу очень сложные вещи про то, как мы считаем. У нас есть статистика по тому, сколько файлов ожидает отправки в MTA. Ok, просто считаем файлы. Статистика супер простая, делается на раз-два.

Второй момент. Число ошибок в MTA. Вы не поверите, но там тоже достаточно просто посчитать файлики, которые были переложены в папочку для ошибочных отправок. Никаких проблем.

Третий момент — это супер сложная статистика, потому что нам пришлось задействовать в ней целый MySQL. Мы стали складывать информацию о том, какой хост сколько отправил писем в MySQL. Грубо говоря, на каждое отправленное письмо он говорит: «О, я еще одно отправил». Было супер сложно, делали, наверное, часов пять. Вася, я прав? По-моему, пять часов.



Как реализовали?

Среднее время отправки в MTA:

- PINBA — наш собственный сервис мониторинга

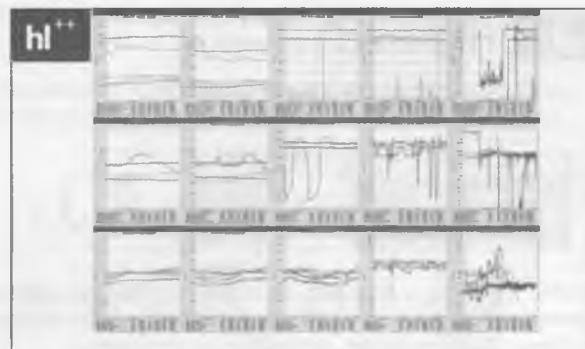
Среднее время доставки почты в отдельный почтовый сервис:

- парсим логи MTA
- хитрая агрегационная структура (highload!!1111)

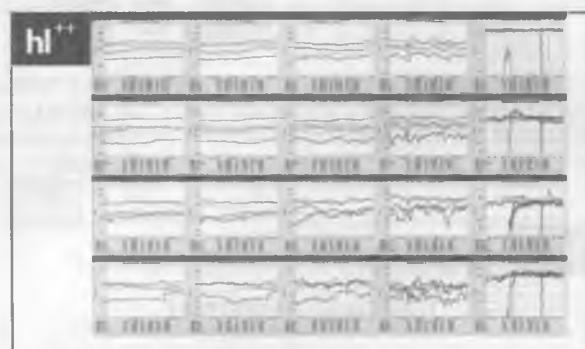
Еще парочка статистик. Здесь уже немного пострашнее.

Во-первых, мы мониторим среднее время отправки письма в MTA. Грубо говоря, сколько выполняется команда «mail», которая уже дергает ssmailto, про который я вкратце рассказал. Померили мы это время просто. У нас есть самописный сервис для обмеров времен чего угодно мониторинга. Называется PINBA. Открытый, доступный — ставьте, пользуйте. Мы им померили команду «mail». Сделали очень страшную вещь — узнали, сколько у нас отправляется одно письмо (по-моему, 150 миллисекунд).

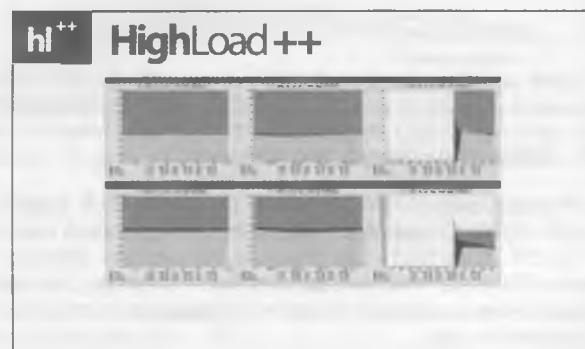
Наконец, последняя статистика. Я буду говорить без юмора. Она прикольная по той простой причине, что нам понадобилось научиться парсить логи почтовых серверов, «Communigate» и Postfix'a. Они делают довольно веселые логи. Чтобы понимать, сколько секунд прошло с того момента, как пришло письмо в наш почтовый сервер, до того, как он отправил его в какой-нибудь Hotmail. Там мы распарсили логи и придумали очень хитрую структуру, которая позволяет нам делать десятки миллионов инсертов и апдейтов в день в одной табличке. Прямо highload highload'ом.



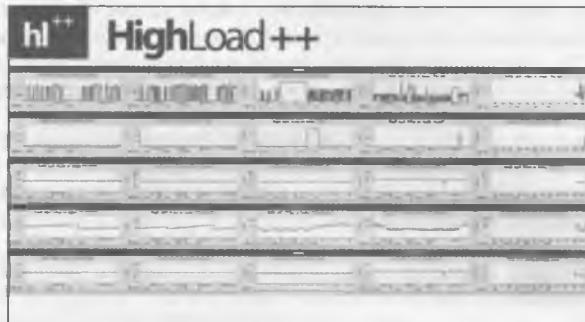
Графики у нас есть такие...



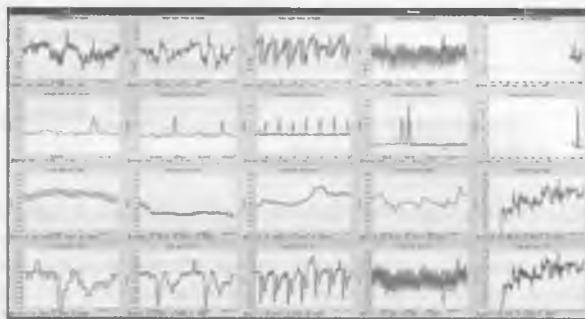
Такие...



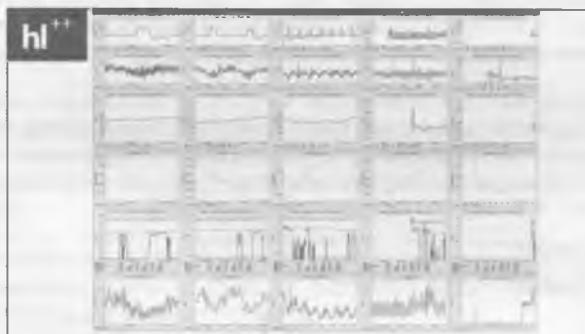
Такие... Это какая-то малая часть.



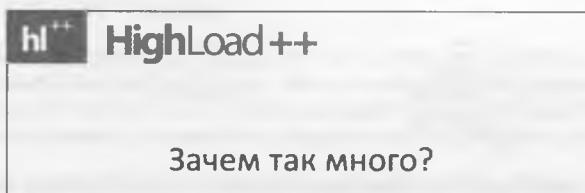
Такие. Это все про почту.



Такие.



Даже такие, похоже. Ладно, немного больше, чем я ожидал. Это какие-то самые основные вещи.



Почему так много получилось? У нас их, на самом деле, раз в пять больше. Почему так много получилось, зачем все это нужно?

Когда к тебе приходят люди с вопросом: «Мы тут письмо хотели, сгенерили, отправили два часа назад, а оно почему-то еще не пришло. Что происходит?», начинаешь понимать, что графиков недостаточно. Ты какую-то новую метрику вводишь, видишь на ней проблему: «Ага, вот на этой машине письма отправляются через два часа после генерации, потому что там этих файлов два миллиона стоит и не успевает сразу отправиться».

После этого ты решаешь проблему и понимаешь, что еще какую-то область не контролируешь, там тоже возникает проблема. Делаешь новый график, понимаешь, в чем проблема, и все потихоньку становится лучше и лучше.

hl++ HighLoad++

Dashboard нас спасёт?

1. Несколько dashboard'ов.
2. Даже dashboard'ы стали слишком сложными.
3. Детектировать аномалии даже на менее значимых графиках автоматически.

Мы сделали много графиков и поняли, что их нужно объединять в dashboard'ы, которых у нас сейчас два. Один dashboard про то, как почта отправляется (там самые важные графики по отправке), и dashboard по приему и обработке почты. Но мы обнаружили, что даже они довольно большие, страшные. На них нужно смотреть, чтобы понять, что сейчас происходит. Когда ты на выходных, ты не смотришь. Когда ты на обеде, ты не смотришь. И вообще, посмотрев два раза в день, остальное время не знаешь, что и как.

У нас, конечно, есть мониторинг, который пишет страшные sms'ки, есть специальные люди (говорят, они существуют), которые звонят, если что-то происходит не так. Тем не менее, есть менее значительные величины, про которые тоже хочется что-то знать.

Следующий шаг, который мы пока еще не сделали, но хотим сделать. Автоматический мониторинг аномалий на графиках, как более, так и менее значимых. Грубо говоря, неделю назад он выглядел так, сегодня повел себя как-то странно. Вчера он выглядел так, сегодня странно. Или, например, в прошлом месяце первую неделю что-то одно показывалось, а сегодня — другое. Что происходит? К этому мы будем потихоньку идти.

hl++ HighLoad++

1984

1. Отправленные:
 - email id
 - IP
 - type
 - language
 - timestamp
 - source host
2. Клики / открытия:
 - email id
 - timestamp
3. Входящие:
 - type
 - timestamp

Я, конечно, не могу сказать, что очень большой поклонник Оруэлла, его произведений. Но должен сказать, что каждый уважающий себя проект должен иметь максимально большой, полный объем информации о том, как вообще все отправляется. У вас должна быть информация о каждом письме, которое вы отправили.

Например, у нас есть информация о том, какой уникальный id'шник был у письма, с какого IP мы его отправили, какого он был типа. Про новые сообщения, про регистрацию, про что-нибудь еще. Язык. На какой машине был сгенерирован, потому что часто проблемы бывают со скриптовыми машинами, генерящими почту. В конце концов, время.

Приблизительно такая же информация по кликам, по открытиям писем. В идеале надо знать, что это конкретное письмо, отправленное тогда-то через такой-то IP, было открыто. Что именно это письмо, а не какое-то другое того же типа, отправленное, тоже было открыто, что-то с ним сделано.

Наконец, у нас имеется информация обо всех входящих письмах. Наверное, пятая часть всего трафика у нас — это входящие письма, bounce'ы, еще много всего. По ним тоже неплохо бы иметь: от кого оно пришло, какого типа было, что и когда.

Эти вещи нам очень и очень сильно помогают каждый раз, когда мы обнаруживаем, что одно письмо из пяти пропало, еще что-то произошло, или нам что-то писали, а мы ничего не получили. Всю эту информацию нужно иметь на руках.

hl⁺⁺ HighLoad++

Выводы

1. Быть гуру не надо, достаточно хотеть разобраться.
2. Правильная архитектура без мониторинга не спасёт.
3. Внезапно: отправка почты – тоже highload!
- 4.
5. PROFIT!!!

Собственно говоря, выводы. У меня доклад не супер громадный. Они заключаются в следующем.

Во-первых. Если вы достаточно большой и разумный проект, чтобы задаться вопросами о том, что вообще с почтой происходит, вам необязательно быть гуру, чтобы эти проблемы успешно найти и решить. Для этого должно быть желание и, конечно, необходимость выделить какие-то ресурсы.

Например, по «Badoo», чтобы отладить, каким-то образом переделать архитектуру, было потрачено, наверное, полтора человека-года. На основные вещи, про которые я здесь рассказывал, требуется значительно меньше времени.

Второй момент. Даже если вы все сделаете замечательно, архитектурно все правильно, пока вы не научитесь это мониторить, смотреть, что происходит внутри, вы никогда не сможете сказать, что у вас все хорошо с почтой.

Третий момент. Я пару раз обращал внимание на то, что это, в общем-то, highload. Да, это highload, я здесь не напрасно рассказываю. Десятки миллионов писем в день — это весьма сложно с вычислительной точки зрения. Если вы все это будете держать в голове и будете готовы решать такие проблемы, то рано или поздно у вас все будет хорошо.

Пользователи будут получать письма в любом объеме вовремя. Маркетинговые программы у вас будут проходить не одну неделю, когда вы говорите своему product team'у: «Ой, ребята, вы знаете, тут вы можете два миллиона писем прибавить, и ваши 50 миллионов мы будем слать 3 с лишним недели». Они к вам приходят, вы им говорите: «Да ладно, за два дня отправим, все замечательно».

hi⁺⁺ **HighLoad⁺⁺**

Ваши вопросы*

* Кроме вопросов о том, как мы доставляемся в Inbox

QUESTION

Собственно, все. Рад буду ответить на ваши вопросы, если такие появились. Если кто-нибудь вообще что-нибудь понял, о чем я говорил. Welcome с вопросами.

Я не пойму, народ то ли одобряет, то ли не одобряет.

Вопросы и Ответы

Вопрос из зала: Как вы узнаете, что письмо открылось?

Андрей Сас: Очень просто. В каждое письмо, которое мы отправляем, вставляем специальную картинку с уникальным кодом. Если пользователь открывает это письмо, и его почтовый сервер или приложение, в котором он открывает, автоматически или каким-то образом подгружает эту картинку. Мы делаем «галочку» в базе и знаем, что письмо данного типа или даже конкретное письмо было открыто. Довольно популярная техника.

Вопрос из зала: (Неразборчиво, без микрофона).

Андрей Сас: Мне говорят, что большинство клиентов блокирует открытие таких картинок. Я не могу сказать, что большинство. Например, Mail.Ru не блокирует. В Mail.Ru все классно. Отправляете, получаете.

Вопрос из зала: Здравствуйте! У вас есть какая-нибудь логика для анализа bounce'ев? Например, у пользователя закончилось место в ящике, и в ближайшие пару дней ему нет смысла посыпать. Тем самым снизить нагрузку на систему.

Андрей Сас: Да, согласен. То есть не согласен. Bounce'ы нужно обрабатывать, на них нужно смотреть. В основном, конечно, мы обрабатываем ошибки типа 550 «Unknown user». Если пользователя нет, писать туда грешно второй раз в ближайшее время.

По поводу того, что у пользователя кончилось место или не кончилось. Нет, мы никак на это не реагируем. Мы просто, естественно, перестаем пересыпал это письмо, поскольку это 500-й код, «Hard bounce». Будет следующее письмо, мы снова попробуем. Это конкретно по вопросу «если кончилось место».

Эти коды довольно редки. На Hotmail, Yahoo, Gmail'ах (основном нашем почтовом трафике) сейчас его редко можно увидеть просто потому, что там большие почтовые ящики. Поэтому это проблема: а) минорная, б) я не считаю, что реально нужно какое-то время не слать. Я же не знаю, когда юзер почистит свою входящую папку.

Вопрос из зала: Второй вопрос по abuse. Есть ли у вас отдельная служба, сколько людей этим занимается?

Андрей Сас: Abuse-служба?

Вопрос из зала: Да.

Андрей Сас: У нас есть саппорт, в который можно всегда написать. Если вы говорите об адресе abuse, на который можно написать письма, да, такая служба есть. Туда приходит очень мало писем. Эта служба — буквально я.

Вопрос из зала: Я имел в виду, что письма не доходят, нужно с этим разбираться. Почему не дошли.

Андрей Сас: Понятно. У нас по этому поводу выстроен специальный процесс, в котором участвует команда саппорта. Грубо говоря, к ней приходит заявка: «Я такой-то пользователь. Вот мой адрес почты. Я трижды заказывал себе письмо восстановления пароля или еще что-нибудь. Пожалуйста, разберитесь, почему оно мне не приходит».

Тут нам очень помогает информация о том, кому что мы слали, какие мы слали письма, заказывал ли пользователь такое письмо, было ли ему оно отправлено. Мы даже видим, открывал он или нет. Когда мы понимаем, что вроде бы письмо послали, открытия не было, проблема может быть. Мы смотрим, большой ли почтовик, на котором такая проблема возникла. Если большой, то мы начинаем с ними связываться, спрашивать, что происходит, попали ли мы куда-то там (естественно, предварительно проанализировав логи почтового сервера, который занимается отправкой). Даем пользователю рекомендации: «Посмотрите в спаме, посмотрите еще где-то».

Да, это все решается через саппорт-команду. Мы перепроверяем тот факт: это проблема юзера, который не очень догадлив, или это действительно проблема с доставкой.

Вопрос из зала: Вы говорили, что используете для хранения писем, а также как временное хранилище файловую систему. Как вы боретесь с фрагментацией, какую файловую систему используете?

Андрей Сас: Никак не боремся.

Вопрос из зала: Не проблема?

Андрей Сас: Да, это не проблема. Никогда с ней не сталкивались.

Вопрос из зала: Скажите, пожалуйста. Вы рассказывали о системе распределения нагрузки по вашим МТА'шным серверам. Не скажете, какую примерно нагрузку разбрасывает железка? Именно количество трафика в секунду. Не по количеству писем, а по объему трафика.

Андрей Сас: Ух, сложный вопрос. Я не знаю. Мы никогда не сталкивались с проблемами на этой железяке. Но насколько я представляю, там миллионы одновременных соединений через нее ходят. Железка, соответственно, денег стоит. Очень веселых.

Вопрос из зала: Понятно.

Андрей Сас: В общем, я про это не могу сказать. С точки зрения почты мы никогда с этим не сталкивались. Где-то в админской переписке я видел что-то про миллионы одновременно держащихся коннектов через эту железку. Она занимается не только почтовыми вещами, а вообще в целом локальным распределением трафика, что, в принципе, следует из названия.

Вопрос из зала: Я знаком с такими железками. Проблем с ней у вас не было? Хорошие отзывы об ее использовании.

Андрей Сас: С точки зрения почты проблем не было. У нас есть специальный админ, который здесь сидит. На нем футболька «Badoo», Вы можете ему этот вопрос задать. Он хоть что-то про это да знает.

Вопрос из зала: Спасибо.

Вопрос из зала: Можете чуть подробнее рассказать, как вы боретесь с тем, что почтовики отклоняют как спам при большом количестве сообщений?

Андрей Сас: К сожалению, закрыли мою презентацию. Да, у меня там было два замечательных слайда, которые, к сожалению, тоже закрыли. Я про это не могу говорить и не буду.

Вопрос из зала: У меня как раз с этим был связан вопрос. Основное узкое место — это принимаемость другой стороной.

Андрей Сас: По поводу всех вопросов с доставляемостью... Вся проблема не в том, какие там почтовики хорошие, плохие. Вся проблема исключительно в том, что вы шлете пользователям. У нас, например, одна жалоба на 400 отправляемых писем. Когда у вас будет такой результат, у вас не будет проблем с доставкой никуда, кроме нескольких тормозных почтовиков Восточной Европы и Латинской Америки из прошлого десятилетия.

Ведущий: Большое спасибо. У нас, к сожалению, закончилось время. Еще можно пообщаться в кулуарах.

Андрей Сас: У меня одно дополнение! Все, кто интересуется отправкой почты и живет в России. Здесь есть специальный человек — Вася (Mail.Ru). Он запускает очень крутой проект. У них можно посмотреть, доставляются ваши письма в инбокс или нет. Какие у вас есть проблемы (или их нет) и как их решить. Я буду стоять на выходе — рядом со мной будет Вася. Я вам его покажу, если вам это интересно. Спасибо.

Высокая нагрузка на erlang-приложения: erlyvideo на гигабитном канале

Максим Лапшин



профилирование Эрливидео

Макс Лапшин
max@erlyvideo.org

Макс Лапшин: ...Проектами видеостримингового сервера Эрливидео. Это серверный софт, который ставится на железку и занимается раздачей потокового видео на все современные клиенты: Flash, iPhone, Android и прочие-прочие.



Эрливидео

- Потоковое видео
- Разные протоколы: MPEG-TS, RTMP,
- Высокая эффективность

Я хочу рассказать про то, как мы адаптировали Эрливидео к большим нагрузкам, когда мы с ними столкнулись. Что из себя представляет Эрливидео в том контексте, в котором мы сейчас будем говорить.

Сейчас будем говорить только про потоковое видео, когда приходит поток, раздается. Как, например, с этой конференции раздается видео прямого эфира на пользователей. Приходится обрабатывать очень много разных протоколов: MREG-TS, RTMP. Огромное количество протоколов. Некоторые из этих протоколов порождают очень высокие нагрузки на сервер. Поэтому нам приходится с ними работать.



Разные нагрузки

- Тысячи пар: один источник, один клиент
- Один источник, тысячи клиентов

Какие бывают профили нагрузки на потоковый сервер. Один профиль нагрузки, с которым сталкивались ребята из «Badoo», когда пытались запустить Эрливидео (то, что мне рассказывали, они делилась), — видеочат. Это характеризуется тем, что у вас много пар соединений. Один источник, один клиент. Это одна история.

Сейчас мы будем говорить немного про другую. Когда у вас один источник, и у него есть тысячи и тысячи клиентов. Условно говоря, когда мы говорим про интернет-телевидение.



Видео

- Один поток: 50 кадров/сек, 500 Кбит/с
- 2000 клиентов — 100 000 кадров/сек.
- Суммарно 1 Гбит/с

Что представляет из себя видео в том контексте, в котором будем рассказывать. В целом в хорошем видеопотоке 25 кадров в секунду, примерно столько же аудиокадров. Итого мы имеем 50 кадров в секунду примерно на один аудио- и видеопоток.

Когда подключено 2 тысячи клиентов, мы получаем около 100 тысяч событий, кадров в секунду, которые надо обработать. Вся беда в том, что этот гигабит видео, который формирует около ста тысяч событий, надо обрабатывать проблемно.



RTMP

- У каждого клиента своё
- Каждому клиенту своя копия потока
- Данные общие, заголовки разные

В чем проблема. Ведь MREG-TS даже Python'ом можно гигабитами раздавать. RTMP — это протокол, по которому подключается Adobe Flash Player. 95-99% обычных пользовательских компьютеров поддерживают RTMP. Вы можете отдать туда видео.

RTMP отличается тем, что для каждого клиента формируется свой собственный поток. К сожалению, вы не можете обойтись простым копированием данных. Необходимо под каждого подключенного клиента сделать свои данные. Я думаю, вы можете догадаться, какие с этим проблемы. Вы не можете сделать одну большую очередь, из которой раздавать всем данные. Под две тысячи клиентов вы генерите две тысячи разных буферов.

Однако есть некоторые облегчения. Можно при некоторых условиях, при некоторых допущениях добиться того, чтобы данные были общими между клиентами, а заголовки разными. Эта проблема порождает большие сложности при программировании всего этого дела.



эрлингвидео

RTMP

- Скорости не хватает — клиент залипает
- Ошибка в тайстемпе — клиент залипает
- Клиент залипает — надо что-то делать

Чем еще характеризуется RTMP. К сожалению, Flash Player — это не VLC. Он написан в Adobe индусами. Без комментариев. Написан он сложно. Работать с ним, действительно, очень сложно. Самое главное — при работе с RTMP-клиентами есть огромная-огромная сложность в мониторинге того, что у вас происходит.

Например, если клиенту немного не хватает скорости, он начинает заливать и недовыгребать данные. Это еще как-то можно контролировать. Но есть и другие прелести.

Например, если у вас произошел скачок времени в потоке. Допустим, камеру подключили — выключили обратно, у вас произошел скачок времени, потому что кодирующая программа начала считать его по-другому. Всё, все клиенты, которые были, залипли. Они перестали делать чтение с сокета.

Эти проблемы очень сложно мониторятся. Со стороны сервера их можно отмониторить только по росту очереди исходящего/входящего соединения у клиента. Надо что-то делать, если вы понимаете, что клиенты не выграбают данные.

Сервер может понять только одну вещь: клиент перестал читать данные по сети. Необходимо каким-то образом догадаться, что с же этим делать, чтобы правильно обработать. Так у вас подключено две тысячи клиентов (две, четыре, больше).

Делать можно разные вещи. Например, можно отключать видеопоток. Можно отключать аудиопоток. Как правило, это не имеет смысла. Если у клиента канал провалился, и он не может выграбить ваше видео, ему вообще нечего делать здесь. Его надо отправить на другой поток с меньшим битрейтом либо просто отказать ему в обслуживании.



Рост эрливиdeo

- Начались проблемы на 800 клиентах
- Неверная архитектура
- Пришлось профилировать

Мы столкнулись со всеми этими проблемами, которые есть у Flash Player, когда начали выкатывать Эрливиdeo на все более и более массовые сайты. Они начались у нас в районе восьмисот клиентов. Эрливиdeo при версии 1.0 мог держать около восьмисот клиентов и все.

Дальше уже возникали проблемы, он просто неправлялся. Люди отключаются, не получают видео равномерно. Нам пришлось профилировать, выяснять, мониторить и учиться профилировать приложения на Erlang. Для нас это было новизной, потому что про это было очень мало данных, очень мало информации.



Профилирование

- Девелопмент одно — продакшн другое
- Разные версии клиентов
- Разные скорости каналов
- Разные задержки на каналах

Какие проблемы при профилировании сервера, который обслуживает один гигабит данных (один, два гигабита и больше). Проблема в том, что вы не можете воспроизвести это на локальной машине. Более того, это сложно воспроизвести у себя в офисе. Это действительно очень сложно. Даже если вы сделаете большой отдел нагрузочных тестировщиков, которые будут что-то имитировать, они все равно будут проверять, что творится у вас в офисе.

Когда вы ставите это на продакшн, запускаете пользователей, начинается совершенно другое. Это разнообразнейшие версии Flash Player'ов, Windows, прочие другие ситуации, с которыми очень сложно бороться.

В целом, конечно, чаще всего все упирается в нюансы с каналами и разными версиями клиентов. Скажем, в одной версии Flash Player'a надо писать одни данные, в другой — другие. Это все очень сложно учесть. Безусловно, финальная стадия профилирования всегда на продакшне. А это большие проблемы.



Профилирование

- Профилирование на продакшне
- Не отключая пользователей
- Без деградации системы

С чем же проблемы. Это не профилирование HTTP-сервера, когда запрос длится 30, 100, 200, 500 миллисекунд. У вас идет подключение, которое может длиться, например, 10 часов, 20 часов или даже неделю. Вам необходимо использовать методики профилирования, которые позволяют не отключать клиентов, во-первых. Во-вторых, без деградации системы.

Если вы хотите понять, что творится при двух тысячах клиентов, когда загрузка на полную, заходите... Чаще всего это делается сервером. Если это приводит к деградации, вы теряете клиентов. Остается 200 клиентов на сервер и все. Вы своими измерениями разрушили измеряемую систему.



Инструменты

- Valgrind и профилировщики тормозят систему
- gdb вносит искажения в код

Когда я делал все то же самое на платформе Objective-C, понятно, Valgrind, профилировщики, gdb были лучшими друзьями. Как без них можно работать с C, я не знаю. Но с ними проблемы.

Например, тот же Valgrind позволяет вам разобраться именно с ошибками. Конечно, он не позволяет разобраться со скоростью. Но Valgrind'ом подключить пять клиентов уже бывает невозможно. Он вносит чудовищные искажения.

Если вам нужно gdb, с ним та же самая история. Вы делаете отладочный build, который во много раз менее быстрый и совершенно по-другому себя ведет.

То же самое с профилировщиками. Все инструменты, которые есть для C++, вносят искажения. С ними проблемы. Что с Java — я, честно говоря, не знаю. Думаю, что тоже не сахар.



Инструменты

- Erlang предлагает другие подходы к профилированию

С чем же я столкнулся, когда мы начали [работать] с Эрливидео. Выяснилось, что в Erlang другие подходы к профилированию, в принципе. Таких инструментов для профилирования, которые есть для C++, в нем просто нет. Но, как выяснилось, они просто особо и не нужны. В них нет такой жесткой необходимости.



Erlang

- Данные и потоки выполнения объединены, как процессы ОС
- Обмен асинхронными сообщениями
- Message box у каждого процесса

Для начала немного про сам Erlang, чтобы вам не было страшно. Что из себя представляет Erlang. Кстати, вон, как раз, вошел дизайнер виртуальной машины Erlang'a.

В Erlang принятая концепция, очень похожая на операционную систему, на сам Unix. Данные и потоки пополнения склеены вместе в одну сущность, очень похожую на процессы операционной системы. Они ровно так же и называются — процессы. Коммуникация между ними происходит не прямым вызовом методов в другом объекте, а посылкой сообщений из одного процесса в другой. Это очень похоже на обычную операционную систему. Это очень удобно.

У каждого процесса есть свой message box, который накапливается и может расти неограниченно.



Erlang

- Все данные немутабельные
- Процесс — рекурсивная функция,зывающая себя с новым значением

В процессах находятся немутабельные данные. У вас даже нет операции поправить какие-то данные.

Сам процесс — это такая рекурсивная функция, которая сама себя вызывает. Вокруг немутабельности данных и изоляции данных внутри одного процесса, который является treadом, построена вся система.



Erlang

- Один процесс под каждого клиента
- Один процесс под каждый «объект»
- Вызовы методов строго последовательны

Какой подход принят в Erlang для обработки клиентов. Под каждого клиента один, два или более процессов. Процесс и объект — это очень похожие по использованию вещи. То, что является объектом в Java, тут будет процессом. Но есть некоторые отличия.

Например, на объекте, который в программе на C++ или в Java, вы можете вызвать метод из двух разных treadов одновременно. Они действительно будут выполняться одновременно. Вам нужны mutex'ы, локи и прочие вещи.

В Erlang такого нет. Каждый процесс обрабатывает входящее сообщение строго последовательно, потому что он не может выполнить два одновременно. Это обозначает, что на всех объектах методы строго синхронны. Они чисто семантически выполняются синхронно.



Lowload

- Процесс-«поток» шлет сообщения процессам-«клиентам»
- Клиенты упаковывают RTMP сами
- в системе FPS*N сообщений

Какая в этой всей базовой архитектуре получается архитектура при малой нагрузке. Под каждого клиента заводится свой процесс. Есть процессы, которые обслуживают сам поток. Это тот процесс, к которому подключается, скажем, видеокамера. Клиенты приходят к потоку, подписываются на видеокадры, получают эти видеокадры в виде сообщений и пишут их в свои сокеты.

Что мы имеем. У нас в системе получается двухтысячекратное дублирование упаковки RTMP-протокола для начала. Во-вторых, у нас получается 100 тысяч сообщений, которые протекают по системе. Несмотря на всю фантастическую эффективность, в систему рассылки сообщений Erlang вложено столько времени, что она безумно эффективна. Но даже ее не хватает на то, чтобы эффективно прокачать по всем пяти, шести, восьми ядрам 100 тысяч сообщений. Получается, что один процесс с одного ядра шлет 100 тысяч сообщений в секунду.

Наверное, в районе 50-ти тысяч сообщений в секунду — это уже около предела того, что реально на среднем ядре может выдать виртуальная машина Erlang'a. В этот предел мы уперлись на восьмистах клиентах. Выяснилось, что процесс, который является потоком, тупо занимается только тем, что рассыпает сообщения. Больше он не делает ничего вообще. Не успевает даже подключить новых клиентов.



Тормоза

- Начинаются «лаги» и «залипание»
- Куда смотреть? Что мерять?

Отсюда получилось, что у нас начинались тормоза, лаги, залипания. Клиент не может даже подключиться к потоку, потому что поток занимается обслуживанием уже имеющихся клиентов. У клиентов начинается плавная, мягкая деградация всего-всего. Как мы выяснили проблему, как мы нашли узкое место.



Мониторинг

- Меряем очереди сообщений
- `erlang:process_info(Pid,message_queue_len)`

Мониторинг — это то, что предлагается в Erlang'e вместо классических систем профилирования. Мониторинг тут спасает как раз из-за других базовых концепций.

Одна из самых эффективных мер мониторинга — это очереди сообщений. Из-за того что все процессы обмениваются только сообщениями, фактически, получается, вызов метода — это асинхронное событие. Мало того что оно асинхронное, в отличие от C++ оно где-то записывается. Message box — это как журнал вызовов к объекту.

Если пришло 10 тысяч процессов, которые от одного процесса что-то попросили, сделали одному вызов, то в очереди этого процесса будет видно 10 тысяч сообщений. Это можно мониторить.

Очень несложным способом мы строим график. Мы смотрим, кто из процессов сейчас загружен, у кого очередь сообщений самая длинная.



Мониторинг

- Меряем загрузку CPU
- etop:start()
- мониторим общую память процессов

Также мы можем мерить отдельно по процессам загрузку CPU. Фактически это означает, что мы знаем, в каком объекте у нас больше всего отъедается процессор. Это очень необычная концепция. Вообще говоря, для C++ слова: «объект отъедает CPU» — это нонсенс. В Java или C++ только тред может отъедать CPU. Но мы не знаем, в каком объекте он сейчас находится, что он сейчас обрабатывает, какую функцию.

Здесь по-другому. Здесь объект и тред — это одно и то же. Мы можем понимать, какой код чем занимается сейчас.



Узкие места

- В узких местах накапливаются сообщения
- Растет CPU, потребляемый процессом
- Накапливается память

Что получается, когда возникают узкие места. С узкими местами возникает рост очереди сообщений. Если один процесс медленно работает, то у него растет message queue. Она может расти неограниченно. 200 тысяч сообщений, миллион сообщений — это бывает. Нормально, в этом нет ничего страшного, это можно разгрести.

Растет его CPU, следовательно. Отсюда мы можем понимать, где у нас накапливается память точно такими же методами, потому что все изолировано.



Узкие места

- Процессу не хватает одного ядра
- Недообрабатывает сообщения
- Накапливается память
- Встают в ожидании клиентские процессы

В какой-то момент возникает проблема, что Erlang прекрасно свои процессы раскидывает по ядрам. 16 ядер совершенно нормально масштабируется. Но один процесс может жить только на одном ядре. Если ему не хватает, мы видим рост. Мы видим, что у него растет очередь сообщений. Клиентские процессы, которые от него что-то хотели, встают в очередь. Они не обрабатываются. Они тоже начинают вылетать по ошибке тайм-аута, потому что в Erlang вызов метода может вылететь и по тайм-ауту. Тем самым мы можем находить узкие места.



Как боролись?

- Уменьшили количество сообщений
- Прямая запись в сокет
- Распараллелили запись в сокеты по ядрам

Что мы сделали в нашем конкретном случае. Мы радикально выкинули рассылку в сотню тысяч сообщений. Передали сокеты вовнутрь процесса потока, который начинает обрабатывать. Он пишет напрямую в RTMP-протокол с сокета. Этот нехитрый, несложный трюк позволил нам увеличить до двух — двух с половиной тысяч клиентов.

Дальше выяснилось, что и этого не хватило. Мы решили сделать хитрее. Мы решили воспользоваться таким трюком. Этот процесс записи в сокеты мы опять же распределили по ядрам. Получается, что процесс и поток — есть около восьми воркеров по количеству ядер. Они держат свой небольшой subset клиентов. Соответственно, поток раскидывает по своим воркерам, а воркер уже пишет в сокеты. Такая несложная схема.

Наверное, многие из здесь сидящих спокойно сделали бы все то же самое на C++, на Java. Вопрос только в том, за сколько времени. У нас на все это ушло четыре дня. Причем из них три дня писал человек, который увидел Erlang позавчера. Я просто объяснил ему, что надо сделать. Он открыл книжку, залез и все сделал, все заработало прекрасно.



Результат

- Во много раз увеличили количество одновременных клиентов

В результате таких наших мониторингов, исследований, профилирования мы смогли радиально увеличить количество обслуживаемых клиентов, в принципе, сняв вообще такую головную боль, как ресурсы процессора на рассылку сокетов. Мы опять уперлись в канал. Понятно, что канал — это самое дорогое сегодня в интернете. Сервера не стоят ничего относительно каналов.



Рекомендации

- Планировать количество проходящих сообщений
- Уменьшать работу внутри вызываемого процесса
- Раскладывать работу по процессам — расползутся по ядрам

К чему был весь этот доклад. Скажем, если вы перестроите свою систему или будете строить новую систему на Erlang'e, вы боитесь того, что у вас могут быть проблемы со скоростью, с задержками, с чем-то еще. Могу сказать: нет, этого бояться не надо.

Есть некоторые рекомендации, которые мы можем дать.

Во-первых. Вам надо немного заранее проектировать, что будет с системой. Прикинули: в потоке 50 кадров, 2 тысячи клиентов, 100 тысяч сообщений. Это очень много.

Во-вторых, конечно, нужно уменьшать работу внутри одного клиента. Например, ситуация с потоком. Клиент может прийти и попросить данные про поток. Скажем, сколько сейчас клиентов подключено к этому потоку. Какие-то еще данные, статистика. Если все эти данные пропихивать с помощью сообщений через штатный механизм, будет проблемно. Это не будет работать, это будет вам вешать систему.

Гораздо эффективнее, чтобы поток сам писал про себя данные куда-то наружу, откуда вы можете читать. Но это совершенно обычные вещи. Если у вас есть какая-то живая система, постарайтесь отказаться от исследования ее в Real Time, от сбора статистики.

Еще что у нас в Erlang'e дается абсолютно даром, без каких-либо проблем — расползание по ядрам. Чем больше процессов вы плодите, чем более гранулярно ваш код разбит по процессам, тем больше шансов, что у вас все будет расползаться по ядрам самостоятельно, соответственно, с увеличением суммарной производительности.

В реальности на Эрливидео виртуальная машина Erlang'a показывает абсолютно линейный рост количества обслуживаемых клиентов относительно ядер. Очень ровно, прямая. У нас нет никаких искажений, нет никаких проблем из-за того, что машина использует 3-4 треда между четырьмя ядрами.



Выводы

- Профилировка приложений на erlang возможен без деградации качества обслуживания
- Методы поиска проблем несложны и эффективны

Что же самое главное мы получили от Erlang во всей этой ситуации. Когда я делал подобную задачу на Objective-C, это был просто кошмар. Отладить подобной сложности сервер с обильной коммуникацией между подключенными клиентами, с тайм-аутами, с утеканием памяти, с прочим и все это без деградации качества, без деградации скорости, без деградации качества обслуживания, без отключения клиентов безумно сложно.

Erlang предлагает нам совершенно другую вещь. Мы можем на ходу, на живом сервере обновлять код, прямо под подключенными клиентами, не отключая их. Если у вас есть какой-то код, который обслуживает клиентов, вы выкатываете новые исходники, они подцепляются, и все работает.

Самое главное, что нам пришлось немного поменять наш привычный набор инструментов, которым мы пользуемся для профилирования, отладки и прочего. Старые инструменты оказались просто нерабочими. Даже их просто нет для Erlang'a. Есть другие, которые при должном использовании оказались гораздо проще и эффективнее, чем те, которыми я пользовался раньше.



Вопросы?

Макс Лапшин

Вопросы и Ответы

Вопрос из зала: Вопрос первый. Я так понимаю, пришлось в некотором смысле пожертвовать модульностью и изоляцией соединений между клиентами. Если раньше RTMP-процесс давал сырье данные всем процессам клиентов, и они занимались упаковкой и подсовыванием нужных конкретному клиенту заголовков, то теперь всем этим сразу занимается один процесс. Правильно?

Макс Лапшин: Совершенно верно.

Вопрос из зала: Не было способа обойтись без этого?

Макс Лапшин: Erlang позволяет воткнуть собственный код обработки сырых сокетов. Можно добраться непосредственно до сокетов в обход той инфраструктуры, которая предлагается Erlang'ом. Можно было бы так. Но, я говорю, мы потратили неделю, оно у нас заработало и перестало болеть и чесаться. Все стало хорошо. Наверное, можно было бы. Если у нас будет проблема с 10-ю или с 20-ю гигабитами, если такое случится, значит, будем решать эти проблемы. Пока у нас нет такой головной боли.

Вопрос из зала: Понятно. Второй вопрос. Казалось бы, еще один общезвестный способ поиска проблемы в перформансе, если не получается воткнуть профайлер на живую, это, например, что логировать. Логирование тоже тормозит?

Макс Лапшин: Конечно.

Вопрос из зала: Настолько сильно?

Макс Лапшин: Что будем логировать? Если будем логировать фреймы, то 100 тысяч записей в секунду не может не нанести никакого искажения. Если мы будем логировать подключения, это очень мало чего даст. Самый важный мониторинг во всем этом — это длинные очереди сообщений в процессах, чтобы понимать, кто чего сейчас выграбляет. Второе — это статус записи в TCP-буфер ядра. Если он перестал записываться, то фактически случилась беда-беда. Это по факту на практике означает, что клиента надо отключить и больше не общаться с ним. Значит, скорее всего, он сидит по модему или плохому интернету. Такая практика.

Вопрос из зала: Можно было бы включить логирование для какого-то процента одновременных клиентов, не для всех.

Макс Лапшин: Что логировать-то?

Вопрос из зала: Я не знаю. Тебе видней.

Макс Лапшин: Собственно говоря, нам хватило банальной интроспекции. Я, может быть, не очень внятно это объяснил. Виртуальная машина Erlang'a позволяет вам зайти в консоль запущенного сервера. Вы можете со своей девелоперской машины попасть внутрь работающего процесса и исследовать все те данные, которые там есть. При этом наличие имеющихся механизмов позволяет вам залезать даже внутрь рабочих процессов и исследовать их состояние.

Конечно, гораздо интереснее и эффективнее оказалось не втыкать логирование. Оно, конечно, полезно, когда ты знаешь, что ищешь. Когда ты не знаешь, что ищешь, просто тупо тормозит, какие-то непонятные отваливания. Клиенты жалуются, все. У нас жалоба: «Ничего не работает». Все пропало. Соответственно, заходим и начинаем смотреть. Если бы нам не хватило недели, мы бы, наверное, уперлись в какое-нибудь логирование, журналирование, переписывание каких-то еще вещей. Самое главное — мы смогли обойтись без профилировщика и без попыток понять, что же он нам говорит такое странное.

Вопрос из зала: Логировать ошибки не пробовал?

Макс Лапшин: Нет, ошибки, конечно, логируются. Но что именно в них?

Вопрос из зала: По ним и определяешь.

Макс Лапшин: Какие ошибки?

Вопрос из зала: Тайм-ауты.

Макс Лапшин: Тайм-аут из-за чего происходит? У тебя же не будет: ошибка тайм-аута из-за того, что неправильно написал такой код. Конечно, тайм-ауты отслеживаются. Особенно учитывая, что в Erlang есть такая концепция: когда один объект вызывает метод на другом, этот вызов может обвалиться по тайм-ауту. Если второй очень долго отвечает, первый отпускает. Ему говорят: «Дружок, твой вызывающий процесс не отвечает». Конечно, этими тайм-аутами лог забит по полной.

Такое логирование не дает тебе ответа, куда смотреть, что тормозит. Такая интроспекция, к которой мы прибегли, позволила очень эффективно найти причину. Узкое место выползает сразу. У кого больше всех отожрано памяти, тот и плохой. После чего мы смотрим, в какой функции сейчас вертится процесс, чем именно он сейчас занимается. Такой поиск может занимать день.

Вопрос из зала: Какое примерно количество нод используется?

Макс Лапшин: Одна. Что касается кластеризации Erlang'a. Я этим никогда не пользовался, не пользуюсь. Мне это не нужно. В Erlang'e есть такая вещь, как кластеризация. Можете поставить рядом два компьютера, запустить на них две виртуальных машины, объединить их вместе. Между ними будет прозрачное сетевое взаимодействие. Это очень похоже на CORBA, только это работает.

Вопрос из зала: На самом деле, я думал задать вопрос на тему сохранения соединений при использовании философии let it crash.

Макс Лапшин: В реальности это не нужно делать. Если вы программируете, например, видеостриминговый сервер и делаете к нему клиент на Flash'e, вы должны очень аккуратно и обстоятельно обрабатывать ситуацию разорванного соединения. Соединение рвется часто.

Вы сидите и смотрите видео. Рядом начал смотреть сосед — все, у вас обоих оборвалось соединение. Сосед закрыл, вы смотрите дальше. В реальности нет смысла так трястись над одним коннектом, как это нужно, скажем, при обслуживании телефонного звонка. Тут — оборвалось, оборвалось. Переконнектились и все. Тем более сейчас что Flash Player, что iPhone умеют обрабатывать разрывы и не обрывать визуально просмотр.

Вопрос из зала: Спасибо.

Макс Лапшин: Оборвалось — и хрен бы с ним.

Вопрос из зала: Был слайд про баги в плеере, в Adobe. Вы с ними столкнулись, начали решать, в итоге пришли к Erlang'y. Не пытались reportить в (неразборчиво) Adobe? Может, это все проще можно было решить?

Макс Лапшин: Конечно же, можно. Но в среднем баг в Adobe решается 2-3 года.

Вопрос из зала: Вы зарепортили или нет?

Макс Лапшин: Я не успею. Так я только к старости денег заработкаю. Суть в том, что Adobe очень плохо исправляет баги. Я сам reportил баги в году 2007 — 2008-м. Сейчас на них начали обращать внимание. Нереально. Просто нет смысла. Либо ты работаешь с Flash Player'ом, либо не работаешь с ним вообще. Если работаешь, то помнишь, какие нюансы есть у каждой минорной версии Flash Player'a.

Вопрос из зала: Такое любопытство: как пришли к Erlang'y? Все-таки экзотика. Был Objective-C. Однажды пришли и сказали: «А давайте на Erlang'e».

Макс Лапшин: Объясню. Я человек ленивый, невнимательный, пишу плохо. Мне хотелось, чтобы были какие-то волшебные гномики, которые решат все инфраструктурные проблемы, связанные с тайм-аутом, утеканием памяти, сборкой мусора и прочим. Какая-нибудь

Java не справлялась. Я видел, как это работает. У нас есть конкуренты, которые написаны на Java. Для них самая классическая проблема — это утекание ресурсов вообще как явление. В Erlang'e тот подход, который есть, который очень похож на сам Unix, практически ставит крест на всем классе проблем утекания ресурсов. Эрливиdeo умеет стоять месяцами и не утекать памятью вообще. Не то что людей нет. Люди ходят, смотрят, работают. Память как влитая.

Я посмотрел. Я думал, какая-то экзотика, какая-нибудь очередная новомодная фигня типа каких-то вещей. Типа Huskey, на который у меня ума не хватает. Оказалось, это очень простой язык. Сам язык реально примитивный. Платформа, которая сделана на нем, чуть-чуть посложнее. Если я беру человека в команду, у меня уходит неделя, чтобы его полностью вовлечь в работу. Когда сам изучал: утром взял книжку и уже к обеду написал рабочий код. Я понял, что это прекрасная вещь. Вся фишка в том, что его делали не академики. Его делали инженеры, которым просто хотелось спать по ночам, а не подниматься и чинить это все.

Вопрос из зала: Еще один вопрос. Вы пишете, что была система, неизвестно где была проблема, вы переписали ее на Erlang, и замечательные средства профайлинга именно Erlang'a показали, что узкое место — это архитектурная особенность самого Erlang'a. У вас большая очередь сообщений. Вы убрали главную архитектурную особенность Erlang'a, и все пошло хорошо.

Макс Лапшин: Это не совсем верно. Есть инструмент. Здесь находится Роберт, который [делал] дизайн Erlang'a. Когда мы ему рассказываем про то, как мы пользуемся Erlang'ом, он слегка удивлен. Они не думали, что он будет так работать, какие-то гигабиты держать. Это вообще не планировалось. Это не то что главная архитектурная особенность.

Главная архитектурная особенность Erlang'a — данные друг от друга изолированы очень жестко. Между данными стоит огромный барьер. Данные слиты воедино с потоком выполнения. Когда вы говорите про Java, что объект вызывает какой-то метод где-то, у вас нет никакой посылки сообщений, никто ничего не делает. На самом деле, у вас [есть] такие безмолвные, бездушные объекты, которые рамазаны по памяти, и есть трэды, которые с ними работают.

В Erlang все гораздо проще и понятнее. В Erlang, когда мы говорим: «Объект посыпает другому сообщение», он реально посыпает другому сообщение или вызывает где-то метод. В этом состоит вся архитектурная особенность. То, что сообщения там — да, мы зашибели виртуальную машину. Выяснилось, что она с этим не работает. Когда вы сталкиваетесь с предельными нагрузками (критичными или запредельными), естественно, у вас часто возникают некоторые нюансы, с которыми приходится иметь дело. Ничего страшного.

Вопрос из зала: Вы сказали, что поток занимался только тем, что разгребал сообщения.

Макс Лапшин: Раздавал, рассыпал. 100 тысяч в секунду — это действительно немало для одного юзера.

Вопрос из зала: Да. На Java 100 тысяч вызовов функции сами по себе не представляют никакой проблемы.

Макс Лапшин: Теперь представьте себе 100 тысяч вызовов через mutex, но в другой поток, которые разгребаются с копированием. При этом без утекания ссылок. С копированием в другую кучу.

Вопрос из зала: Если у меня 100 тысяч ядер...

Макс Лапшин: Нет, какие 100 тысяч ядер. Давайте говорить реально. Я на Марсе ничего не выкатываю. У меня нет там data-центров. Если у вас есть — хорошо. Я пока компьютеров со ста тысячами ядер не видел. Обычно 2, 4, 8, если говорить про реальную жизнь.

Вопрос из зала: Это значит, что 100 тысяч вызовов с mutex'ом занести, я что-то совсем не то сделал в архитектуре.

Макс Лапшин: О чём и речь. Я про это рассказываю. Мы ошиблись в архитектуре. Когда мы пошли искать ошибку в архитектуре (это вообще сложная задача), мы нашли ее моментально. Она была исправлена силами неквалифицированного специалиста.

Речь идет про то, что поиск таких ошибок в Erlang — это очень легкая вещь. Как искать в Java то, что у вас там где-то код реально упирается в количество mutex'ов, я думаю, что это несладко. Это будет явно не силами человека, который стоит вам 20 тысяч рублей в месяц.

Вопрос из зала: Какой оверхед на посылку сообщений?

Макс Лапшин: Оверхед сначала может показаться немножко большим, потому что кучи в Erlang'e разделены. У каждого процесса своя собственная куча. Это ровная область памяти, непрерывная. Это у каждого процесса. Соответственно, в ней идет вся локация. Когда один процесс другому посыпает данные, он сначала накладывает mutex на кучу, на конец кучи того процесса. При этом записывает туда данные, потом отпускает и уже под другим mutex'ом его информирует, что у тебя новое сообщение. На практике это не так страшно. Это работает нормально. Сам по себе оверхед можно простить за то, что ровно загружены ядра. Он есть, но он не фатальный. Он компенсируется идеальным масштабированием по ядрам.

Вопрос из зала: Спасибо.

Ведущий: Спасибо большое. У нас закончилось время доклада. С Максимом еще можно будет поговорить в кулуарах, я думаю.

Специализированные http-демона: круг решаемых задач, подходы и метод

Сергей Боченков, Александр Панков

Специализированные http-демона



- Качественное решение вопроса производительности;
- Высокая стоимость разработки;
- Особенности эксплуатации;

Александр Панков: Дорогие коллеги! Речь пойдет о высокоскоростных технологиях. Вы видите единственный рисунок в этом докладе — это ракета. Мы не проектируем сверхзвуковые самолеты. Мы занимаемся только высокоскоростными вещами.

Речь пойдет о написании так называемых специализированных демонов. Когда они пишутся? Когда уже не обойтись классическими решениями либо классические решения обходятся слишком дорого. Специализированные демоны могут позволить даже не каждая богатая компания, потому что для этого нужны соответствующие высококвалифицированные специалисты.

Я представляю систему Advaction (нашу рекламу вы можете посмотреть в интернете). Со мной будет выступать и говорить конкретно о технологии Сергей Боченков, эксперт в области высокоскоростных технологий. Высокоскоростные — это десятки тысяч запросов в секунду, а не просто тысячи.

Когда нужны и когда нет?

Нет

- Менее 1000 запросов в секунду;
- Работает неплохо на стандартных решениях;
- Нет уверенности в квалификации специалистов;

ДА!

- Принципиально решить вопрос производительности;
- Есть время и деньги на разработку;

Вы должны определиться для себя: нужны ли вам настоящие высокоскоростные технологии. Если у вас мало нагрузки, если у вас уже работает текущее решение, вам стоит задуматься,

нужно ли вам переходить на какие-то свои разработки. Если у вас нет команды, способной решить эти вопросы, вам тоже стоит задуматься, может быть, вы можете отмасштабировать ваш «Битрикс» на много серверов, и у вас все будет работать.

Тем не менее, если у вас есть и возможность, и кадры, и все, качественно решить вопросы производительности можно всегда.

Где используются?

- Системы статистики (счётчики);
- Рекламные системы;
- Внутренние модули посещаемых сайтов;
- Чаты, мессенджеры на крупных сайтах;

На слайде приведены совершенно простые примеры. То, что вы видите в интернете каждый день, и где вы видите специализированные демона. Счетчики: TOP Mail.Ru, Live Internet, Google Analytics, Яндекс.Метрики. Рекламные системы практически все самописные, начиная от баннерных систем 1998-го, 1999-го года, весь «внутрь» высокопосещаемых сайтов очень часто самописный. Чаты, мессенджеры — это все тоже часто пишется на каких-то внутренних, своих собственных разработках.

Я немного расскажу о тех разработках, которые используете мы. Эти разработки частично вились мной, частично командой, которую очень долго приходилось собирать из самых классных специалистов Рунета по высокоскоростным технологиям.

Вопросы архитектуры

Архитектура определяется задачей, чаще всего язык C или C++.

- Однопоточная или многопоточная модель;
- Принцип работы с сетевыми соединениями;
- Необходимость работы с ФС;
- Необходимость выполнения фоновых задач;
- Общие RW-данные, вопросы блокировок;
- Вопросы старта и завершения, непредвиденные падения;

Я совсем забыл рассказать: на слайде круг вопросов, на которые стоит обратить внимание, когда вы начинаете проектировать самописные демона.

Кстати, забыл задать вопрос. (Обращаясь к слушателям). Кто писал какие-нибудь, хотя бы самые простые демона на Perl'е или хотя бы на C++? (Ответы из зала). Отлично. Хотя бы кто-то есть. Значит, кому-то это будет интересно и полезно услышать.

Вопросы архитектуры, в принципе, очевидны и понятны. Если у вас уже есть рабочий прототип системы, нужно понять, что вас в ней не устраивает. Либо у вас не ладится с диском, либо отъедается много памяти. Этот тонкий момент и устраняется с помощью написания именно решения, заточенного под конкретную задачу.

Однопоточный демон wz-htpd

<http://github.com/bachan/wz-htpd>

- Простейший однопоточный сервер, копипаст 0w;
- Подключаемые со-модули;
- Простота разработки модулей;
- Ответы не длиннее 32К;
- Моментальный ответ, никаких ожиданий;
- Допускается создание фоновых потоков, почти не блокирующих основной процесс;
- kevent/FreeBSD + epoll/Linux;
- Нежелательна работа с диском при обработке запросов;

Первое, о чем я хочу рассказать, это однопоточный демон, который я начинал писать еще в далеком 2005-м году, когда работал в Mail.Ru. На самой первой версии этого демона до сих пор работает счетчик TOP Mail.Ru. За основу был взять сервер 0w Максима Зотова. В узких кругах это очень известный вер-сервер для отдачи статики.

Что хотелось добиться. Сервер wz позволяет быстро (в течение 10-15 минут) написать свой собственный веб-сервер для обслуживания коротких, быстрых, моментальных запросов, которые не обращаются ни к диску, ни к каким-либо базам, с короткими и понятными ответами. Чаты, мессенджеры, счетчики, какие-то узловые конечные части других крупных систем.

Со временем разработка росла. Сейчас демон, в принципе, готов к использованию. На слайде показана ссылка, вы можете его скачать, посмотреть. Для реализации достаточно отнестись от небольшого классика и реализовать функцию handle с двумя простыми переменными — что на вход, что на выход.

Простой демон работает в пяти или шести компаниях, вполне успешно внедрялся. Вы можете его попробовать. Он незатейлив, примитивен, но крайне быстр, и под него легко писать модули. Приноровившись, можно очень быстро и легко решать конкретные задачи. Например, замена memcached.

Многопоточный демон blizzard

<http://github.com/bachan/blizzard>

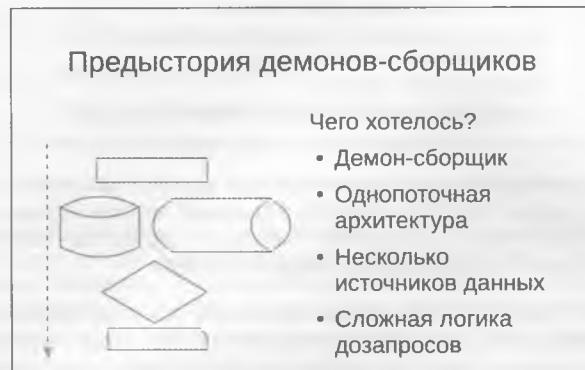
Проектировался для обработки запросов, которые могут обращаться к диску.

- Подключаемые со-модули;
- Два набора потоков — быстрые и медленные;
- Быстрые запросы — моментальный ответ, обращение к диску не допускается;
- Медленные запросы могут обращаться к диску или иному хранилищу данных;

Следующий демон, который хочется представить. Вы тоже можете попробовать его использовать для решения ваших задач. Он тоже узкоспециализированный, но предназначен для обработки двух запросов, которые могут быть короткими и длинными (в плане того, что лезут или не лезут на диск).

Что в этом демоне хорошо. Там два пула потоков. Один занимается обслуживанием быстрых запросов, один — медленных запросов. Если быстрый отвечает, что он не знает, как ответить на запрос, запрос передается в медленный. В этом случае при старте либо при каких-то тормозах системы, если тормозят медленные запросы, то это никак не затрагивает производительность быстрых запросов.

Типичный пример — обращение к файловой системе. Если у вас файлик уже считан в память, вы можете ответить на него из внутреннего кэша демона. Если нет — отправить в длинный тред, и он зачитается с диска и перейдет.



Следующая интересная штука, о которой я хочу рассказать (собственно, ей и посвящен доклад). Родилась идея так называемого демона-сборщика. Что хотелось здесь сделать. Написать некий http-демон, который на вход получает запрос (как неудивительно, http) и подряд делает серию http-запросов, исходя из их ответов, делает другие запросы.

На слайде нарисована незатейливая схема. Исходный запрос, дальше мы, допустим, два раза подходим, какую-нибудь базу спрашиваем. Узнав что-то из базы, лезем в третье хранилище. Потом генерируем ответ в виде JSON'a либо на каком-то шаблонизаторе отвечаем http-ответом.

Как обычно решаются такие задачи. На PHP пишется самое простое решение — «сделать запрос», «запрос в базу». В этом случае один запрос (к примеру, пусть он выполняется пол-секунды) будет занимать минимум один поток или один процесс, что, как минимум, сильно расходует память. Если у вас одновременно 1 тысяча висящих запросов, то вам надо одновременно 1 тысячу умножить на n мегабайт «оперативки». Хочется делать это однопоточно, быстро. Хочется такой демон.

Учитываем, что мы хотим здесь сложную логику. К примеру, 5-10 источников данных, сложная схема (когда куда ходить, когда куда неходить). Как мы пробовали решать эту задачу.

Демона-сборщики на nginx

Плюсы

- Решают задачу;
- Простая эксплуатация;

Минусы

- Сложно разрабатывать;
- Ещё сложнее корректировать логику запросов;
- Отсутствие so-модулей;
- Отсутствие запросов класса «отправил и забыл».

Первое, что мы попробовали, это написать демона-сборщики на nginx. Сережа, кстати, очень хороший эксперт по nginx, он знает всю внутреннюю архитектуру и очень хорошо ладит с nginx'овыми сабреквестами. Он сейчас расскажет, как все-таки тяжело разрабатывать модули для nginx.

Сергей Боченков: Мы попробовали разрабатывать. Плюсы были такие. У нас получилось написать несколько демонов-сборщиков в качестве модуля nginx. Они все работали хорошо и до сих пор работают.

Но минусы были в том, что написать их мог только я почему-то. (Смех в зале). Хотелось бы, чтобы сборщики с такой логикой, которая была указана на предыдущем слайде (с совершенно простым описанием), транспортировались в текст на «Сях» и его мог написать какой угодно программист. В nginx это сложно. Мы захотели написать такой демон, который мог бы так сделать.

Требования к новому демону

- Сравнимые с модулями nginx по производительности и требованиям к ресурсам;
- Простота разработки модулей;
- Модули могут не дожидаться ответов бекендов;

Александр Панков: Я хочу уточнить. Когда Сережа был недоступен, мне очень хотелось изменить логику в одном из модулей nginx. В принципе, я все сделал, но вместо одного ответа он выдавал два. Он должен был выдавать empty_gif, а выдавал два empty_gif. Nginx настолько сложно устроен, что вырезать ненужный кусок body было достаточно сложно.

Поэтому мы подумали, и я решил, что нужно писать что-то свое, универсальное, чтобы пункт «простота разработки» стал центральным. Мы хотели сделать что-то такое, что бы позволило легко и просто писать однопоточные мультиплексирующие демона, чтобы я, не дергая Сережу, мог исправить логику, что-то дописать, подправить. Чтобы это было действительно просто писать. Получилась такая архитектура.

Архитектура демона ugh

- Не отдаляться от зарекомендовавших себя ключевых идей демона nginx;
- Однопоточный http-демон;
- Работа с сетью через libev;
- Подгружаемые so-модули;
- Переключение между контекстами осуществляется с помощью корутин;

Расскажи, почему ты взял за основу nginx.

Сергей Боченков: На слайде перечислены пять основных пунктов архитектуры написанного демона. Первое. Поскольку я достаточно хорошо знал nginx (это отличный веб-сервер, там было много хороших идей), я решил все хорошие идеи, которые мне подходят, взять из nginx. Я буквально копировал какой-то код, когда писал это.

Кроме того, демон ugh, который у нас получился, однопоточный. Работа с сетью происходит через libev (замечательная библиотека для работы с событиями). Он поддерживает подгружаемые so-модули, что нам тоже очень удобно. Все остальные серверы у нас именно так и работают. Не надо компилировать для нового модуля весь сервер заново. Просто подгружаешь модуль — и все.

Простота написания модуля организована с помощью механизма корутин. Про него будет на следующем слайде.

Кратко о корутинах на примере

```
ugh_subreq_t *r = ugh_subreq_add(
    c, url, url_size UGH_SUBREQ_WAIT);
ugh_subreq_set_body(r, body, body_size);
ugh_subreq_set_header(r, ...);
ugh_subreq_set_timeout(r, 0.1);
ugh_subreq_run(r);
ugh_subreq_wait(c); /* switch coroutine */
```

Здесь не все обязательно писать. На самом деле, все не так страшно. Механизм корутин. Корутины — это такие штуки, которые внутри одного и того же процесса, несколько контекстов выполнения, каждый со своим стеком. Между ними можно быстро переключаться. Это дешевле, чем переключаться между тредами.

С помощью механизма корутин была сделана такая штука. Мы можем создать несколько подзапросов, с помощью такого вызова мы можем указать на подзапрос, задать для него какие-то опции (это все необязательно), потом сказать, что мы готовы его выполнять. Мы можем сказать так сколько угодно раз. Потом мы можем вызвать эту функцию ugh_subreq_wait. Она переключит контекст на корутин, который исполняет event loop.

Потом она вернется сюда, и мы сможем продолжить исполнение модуля, только когда ответы на все подзапросы, которые мы здесь заказали, вернутся нам. Мы здесь их сможем прочитать, как-то использовать, опять сделать кучу подзапросов и опять вызвать эту функцию (если хотим). Если не хотим, можем не вызывать. Если мы не поставим этот флагок, она не будет ждаться.

Александр Панков: К сожалению, на слайде очень сложно отобразить это все. Если внешне посмотреть на исходники модуля, написанного для демона `ugh`, он выглядит так, как будто он многопоточный. Вы пишете: «Сделать такой-то запрос, сделать пачку таких-то запросов». Все это выглядит, на самом деле, последовательно.

Просто иногда встречается функция `ugh_subreq_wait`, которая просто-напросто переключает контекст. В этом время корутина временно прекращает выполняться.

Механизм корутин, кстати говоря, почему-то непопулярен, редко используется. Незаслуженно ему не уделяется внимание. В крупных компаниях (таких, как Яндекс) они часто используются. Сейчас с корутиными стало очень просто работать благодаря нашему демону именно для написания таких http-серверов. Внешне все выглядит как многопоточный код. Любой программист, который может хорошо линейно проследить логику, написать на «Сях», в принципе, не способен даже ошибиться, кардинально испортить работу демона и поменять логику.

Демон `ugh`

Исходники и пример использования:

<http://github.com/bachan/ugh>

Самое интересное. Исходники `ugh` можно взять, посмотреть, попробовать, там есть пример. Скомпилируйте его, он должен заработать. Несмотря на то, что синтаксис `ugh` в `nginx`'овом стиле... С-шный код обычно не такой простой на вид, как «плюсовый» код. В «плюсовом» есть объекты, через точку методы вызываются.

Здесь же, по сути дела, это объектно-ориентированное программирование на С. Но ничего не мешает написать so-модуль, который бы вызывал какой-то ваш «плюсовый» класс с помощью тех же самых функций. Из ++ в С переключиться легко.

Настоящее и будущее `ugh`

Сегодня

- Работающая альфа-версия;
- 2 типа запросов: с ожиданием и без ожидания ответа;
- Возможность сделать несколько запросов параллельно;
- Некоторые плюшки `nginx` (переменные, мапы, апстримы и др.);

Завтра

- Развитие возможностей конфигурирования, директива `location`;
- Запросы в БД (`mysql`, `pgsql` и другие) и другие типы источников (`fastcgi`);
- Язык высокого уровня для написания скриптов (например язык Рамблер.NL :));
- Больше плюшек;

Немного о том, что из себя представляет `ugh` сейчас. Это альфа-версия. Но несмотря на это она, в принципе, работает хорошо, и работает на продакшне. Работает под нагрузкой. Син-

тетические тесты она проходит идеально. Этот демон можно пробовать и применять в своих проектах.

Отдельно хочется упомянуть, что нами реализован внутри механизм запросов. Так называемые запросы, ответ которых нам неважен. Например, какие-то статистические запросы. Мы хотим дернуть какую-то ссылочку, но не хотим дожидаться ответа от нее. Просто про нее забыть. Дернуть — и нам неважно, что там ответят. Скорее всего, там ответится `empty.gif` или пустой ответ.

Так реализуются всякие счетчики. Они очень быстро отдают `empty.gif`, пиксель 1x1. Куда дальше пойдет запрос — уже неважно. Главное, что демон в однопоточном режиме моментально (за 5 миллисекунд максимум) отвечает `empty.gif`’ом.

Параллельность запросов — тоже немаловажная вещь. Например, можно зайди параллельно в несколько баз данных. Например, список постов на странице с комментами. У всех есть ники. Берем пачку запросов, отправляем и узнаем профили всех людей в параллельном режиме. Все выдернули — и дальше пользуемся данными.

Как будет развиваться `ugh` — неизвестно. На самом деле, уже сейчас понятно, что хочется. Хочется, чтобы в этом демоне появилось как можно больше фишек `nginx`’а, удобных для админов. Уже сейчас есть все переменные, можно задавать апстримы, апстримы с тайм-аутами, маски есть. Есть совершенно потрясающая вещь — можно написать свою хеш-функцию и расшарить на несколько серверов, и запрос пойдет по апстриму по нужному `location`.

Хочется запросы к базе данных. Это очень легко реализовать. Чтобы можно задать в `ugh`: «Сделай мне SQL-запрос в MySQL». К сожалению, сейчас это нельзя сделать, потому что `slave` клиент в MySQL не предназначен для работы с событийными машинами. По всей видимости, мы напишем свой `slave` клиент напишем, потому что MySQL-исходники корявые.

Вопрос из зала: (Неразборчиво, без микрофона).

Александр Панков: Да, можно, но это надо реализовать. (Смеется). Если бы кто-нибудь взялся реализовать. Просто надо время.

Протокол там ясный и понятный. О протоколе MySQL мы завтра расскажем поподробнее. У нас будет другой доклад, тоже интересный, как раз про MySQL. Там мы тоже затронем темы клиентской библиотеки.

Еще хочется делать запросы `fastcgi`. Это совсем просто, но не реализовано.

Вопрос из зала: Мультиплексирование?

Александр Панков: Мультиплексирование уже есть внутри.

Сергей Боченков: Естественно, если мы будем делать поддержку других типов `back-end`’ов, и все будут по той же схеме. Можно запустить несколько параллельно и все подождать (или какие-то не ждать). Все это — общая схема. Но сейчас она только для `http` реализована.

Александр Панков: Еще мне хочется прикрутить язык высокого уровня — JS, PHP или Rambler NL (новый язык). Скоро о нем будет известно всем, наверное. В Rambler’е рождается новый язык, очень хорошо предназначенный для написания веба. Скорее всего, в `ugh` в качестве модуля мы встроим поддержку в том числе этого языка.

Какие здесь сложности. Допустим, для реализации PHP, в принципе, ничего сложного нет. Но надо описать все API (а оно достаточно богатое у PHP) из работы с MySQL и другими базами данных. Все это реализовать не так-то просто.

Вот такой демон у нас сейчас есть. Можете смотреть, пользоваться.

Спасибо за внимание!

Сергей Боченков (bachan@advaction.ru)
Панков Александр (pianist@advaction.ru)
Компания Advaction

Спасибо за внимание. Можете задавать вопросы.

Вопросы и Ответы

Вопрос из зала: Вы не боитесь, что через некоторое время разработки у вас получится тот же самый nginx с дополнительным набором модулей? Например, в nginx можно встроить Lua.

Сергей Боченков: Я до сих пор не умею на nginx делать такие красивые параллельные запросы. Фишка была в том, чтобы очень просто написать модуль, который делает вообще какое-то безумие.

Вопрос из зала: Killer фича, которая отличается вас от nginx, это только это?

Сергей Боченков: Можно сказать, что только это. Но при этом мы никогда не будем уметь раздавать файлы. Это не нужно. Это отлично умеет nginx.

Вопрос из зала: Ваш коллега, который тоже уволился из Mail.Ru, написал CAS. По докладу, я так понял, что тот же самый функционал. Вы что-нибудь слышали о CAS? Извините, но вы как-то похоже.

(Смех в зале).

Александр Панков: Дело в том, что CAS предназначен, скорее, для обработки сотен и тысяч запросов в секунду. Этот сервер многопоточный. Это сервер для сложной бизнес-логики. Это несколько другой сервер. Он многопоточный. Этот — однопоточный.

Если CAS делает subrequest, MySQL Query, он блокирует поток. В это время вся память, выделенная на контекст решения этой задачи, просто-напросто пристаивает. Демон на корутинах крайне экономичен в этом плане. Во-первых, корутины быстрее переключают контексты, чем треды (это очевидно всем). Корутины несут меньше накладных расходов по памяти. Они фактически только делают копию стека, и то — сколько надо.

Сравнивать CAS и ugh некорректно. Ugh предназначен для десятков тысяч запросов в секунду, когда CAS уже не будет справляться.

Вопрос из зала: По поводу комментариев к CAS'у. Человек, который уволился, это я. Произошло это давно — 6 лет назад, поэтому говорить о том, что оттуда все увольняются, несколько неправильно. В любой большой компании кто-то рано или поздно уходит, приходит. Неважно.

Да, это два совершенно разных продукта. Например, на этой штуке, как я понял из предварительного общения, действительно нельзя рассчитывать какую-то бизнес-логику, которая требует вычислений, времени, еще чего-нибудь. Это вещи из разных вселенных. Если здесь требуется что-то быстро сделать и отдать в один поток, то в CAS можно считать рендеринг картинки 3 на 4 тысячи пикселей — и все будет нормально, потому что это делается в отдельном потоке. Соответственно, сравнивать их несколько неправильно. Это действительно разные вещи.

Александр Панков: Это так же, как сравнивать Apache и nginx. Nginx предназначен для отдачи статики и проксирования. Apache предназначен для богатого функционала и сложного конфигурирования, вплоть до Java-апплетов.

Вопрос из зала: Вопрос к Сергею. Почему выбрана библиотека libev, а не libevent?

Сергей Боченков: Потому что знал ее лучше. У нее лучше результат benchmark'ов, насколько я знаю. Я всегда пользовался ей.

Вопрос из зала: Во-первых, в libevent'е есть уже готовый http-сервер. Он писал такой же демон. Он выдерживает 3 тысячи запросов. Счетчик кликов для баннерной системы сразу с редиректами, с несложной логикой. Там можно даже писать либо в базу, либо в MongoDB (как я писал). Все реально. Узкий сервер под узкую задачу. Я сделал однопоточный, но собираюсь сделать многопоточный сервер. На libevent можно хорошо работать.

Сергей Боченков: Я знаю, мы использовали эту штуку. Но мне удобнее пользоваться libev.

Вопрос из зала: Здравствуйте! Меня зовут Павел. Вы хотите вкручивать MySQL, высокоуровневый язык. Есть такой Tornado веб-сервер. Он написан на Python'е поверх E-pool'a и выдерживает 9 тысяч на синтетических тестах. Там есть все из коробки — Python с батарейками. Почему вы не пробовали его?

Александр Панков: 9 тысяч запросов синтетических тестов с local host?

Вопрос из зала: Hello world, да. Один поток — 9 тысяч.

Александр Панков: Если было бы 90 тысяч хотя бы — тогда можно было бы о чем-то разговаривать.

Вопрос из зала: Зато там Python и все lib'ы. Вы напишете то же самое — будет так же медленно.

Александр Панков: Я опять же хочу вернуться к первому слайду.



У нас нет цели использовать какую-то технологию.

Вопрос из зала: Ты сказал, что по сравнению с тредами по памяти оверхед в корутинах будет меньше. Не совсем понятно, откуда взялся этот меньший оверхед. На мой взгляд, оверхед там будет ровно такой же, как и с тредами.

Александр Панков: На самом деле, на стек можно меньше выделять памяти. Конечно, в тредах можно отконфигурировать. Во-первых, в корутинах ты сам контролируешь переключение, когда оно происходит. В тредах ты этого не контролируешь. Во-вторых, оно дороже.

Вопрос из зала: Дороже что — память?

Александр Панков: Переключение.

Вопрос из зала: С переключениями я согласен. Процессор в корутинах будет использоваться меньше, но я не вижу, где получается выигрыш по памяти.

Александр Панков: В принципе, замечание корректно. Наверное, потому, что я в своей жизни никогда не конфигурировал память под стек для треда. Как получалось, так и получалось. Наверное, если задаться целью, то и для тредов можно добиться примерно того же самого.

Вопрос из зала: Но в тредах память сейчас виртуальная. На 64-хбитных машинах ты просто расходуешь адресное пространство. Физическая память ровно та же самая, что и с корутиными.

Александр Панков: В принципе, так. Надо пробовать, надо тестировать. Идея использования корутина родилась, прежде всего, у меня. Мне всегда не нравилось многопоточное программирование. Вопрос элементарных блокировок. В демоне `ugh` не надо ничего блокировать. К примеру, у тебя есть внутренний маленьких кэш, ты всегда знаешь, что ты обращаешься к нему один.

Если у тебя появляется многопоточный, у тебя сразу `mutex`'ы, как минимум, либо какие-то другие технологии синхронизации тредов. Это тоже накладные расходы. Какие-то общие данные для всех тредов, да еще и не в `read only` варианте, вполне возможны. Тут корутины дадут именно отсутствие блокировок.

Как мы храним 75 млн пользователей (пишем неблокируемый сервер)

Денис Бирюков



HighLoad++

Как мы храним 75 млн пользователей (пишем неблокируемый сервер)

Бирюков Денис
Компания Каванга

Денис Бирюков: Здравствуйте! Продолжаем тему больших нагрузок. Правда, у нас сервер более простой — не http.

Как мы храним 75 миллионов пользователей. Мы пишем свой неблокируемый сервер.



HighLoad++

Самопиар :)



- Рекламная сеть (много баннеров, много сайтов)
- При показе баннера мы используем таргетинг
 - уникальные ограничения
 - ретаргетинг
 - таргетинг по соцдему
- Нужен сервер
 - Помним кому, где и что показывали
 - Знаем пол и возраст
 - Решаем что именно показать

Немного самопиара. Мы — рекламная сеть. Мы должны крутить много баннеров на довольно большом количестве сайтов. При показе баннеров мы должны учитывать какие-то таргетинги, чтобы эти баннеры эффективно отдавать.

Среди всех таргетингов можно выделить группу таргетингов, использующих информацию о пользователе, который пришел за баннером.

Можно разбить эту группу таргетингов на уникальное ограничение, ретаргетинг и таргетинг по соцдему.

Уникальное ограничение. Показывать пользователю «не больше чем 5 раз всего», «не чаще чем 3 раза за 10 минут» или «до первого клика».

Ретаргетинг. Если пользователь был на каком-то сайте — например, svyaznoy.ru, то ему показываем рекламу «Связного», а другим не показываем.

Таргетинг по соцдему. Соцдем — это, например, «мужчина от 25-ти до 30-ти лет». Вот этим мужчинам показывать, остальным не показывать.

Для этого нам нужен целый сервер, который помнит, кому, где и что показывали. Он знает этот пол и возраст и решает с ретаргетингом, что именно показать.

Такой сервер у нас был.

HighLoad++

Что было?

Многопоточный сервер

<ul style="list-style-type: none">• - Блокировки• - Фрагментация• - Низкий КПД по памяти (в 3,5 раза) удаление объектов раз в час	<ul style="list-style-type: none">• + 5% CPU• + Высокая скорость ответа• + Сохранение на диск без доп. памяти• + Быстрый запуск
---	--

Что он из себя представлял? Это был многопоточный сервер со всеми своими плюсами и минусами. Так как он многопоточный (а данные в единственном роде хранятся), то были блокировки, была фрагментация. Аллокаторы не использовались и не писались ни для контейнеров, ни для операторов new/delete.

Нужно было вводить блокировки. Это узкое горлышко — чем меньше, тем лучше. Фрагментация и тот факт, что аллокаторы для стандартных контейнеров не писались, приводили к тому, что был довольно низкий КПД по памяти.

Предположим, что у нас есть сервер. Возьмем всю память, которую он «съел» в текущий момент времени. Выкинем из этой памяти служебные куски, которые тратятся, например, на контейнеры. Принимая во внимание фрагментацию, запишем все, что получилось, на диск. Мы получим сжатые данные без служебной информации, которые занимают объем в 3,5 раза меньше того, что был в памяти.

Удаление объектов производилось раз в час. Например, за час к нам могло прийти очень много пользователей — внезапный всплеск.

Что делать, когда мы исчерпали доступные нам лимиты по памяти? Либо мы в swap уйдем и окончательно ляжем, либо мы будем игнорировать пакеты, которые должны создавать какие-то объекты в системе. И то, и другое плохо.

Однако у этого сервера были преимущества. При отсутствии записи на диск он потреблял в среднем около 5% CPU. У него была довольно высокая скорость ответа. Он сохранял на диск без выделения дополнительной памяти. Для этого отдельный поток бегал по памяти, искал информацию о пользователях, писал на диск. Ничего дополнительного не «кушали».

У сервера был быстрый запуск. Он открывал сокеты на accept, был готов принимать соединение, обрабатывал их. Параллельным treadом он читал базу (если такая была) с диска и забивал свою память пользователями.

hl++ HighLoad++

Задача

- 75 млн пользователей за 3 МЕС, в день меняется 250 млн объектов их описывающих (~ 60 ГБ памяти)
- До 3000 запросов/сек на обновление данных и столько же на их выборку
- Время ответа критично (<= 200 микросек)
- Сервис должен быть масштабируемым

Нам нужно было его переписать. Надо было избавиться от минусов, постараться сохранить плюсы. Что можно сделать?

Задача. 75 миллионов пользователей за 3 месяца (это согласно данным статистики за какой-то период). В день меняется примерно 250 миллионов объектов, которые описывают этих пользователей. Нам это стоит приблизительно (оценить тяжело, помню цифру 3,5 раза) 60 ГБ памяти. В пике приходится до 3000 запросов в секунду на обновление и столько же на выборку данных.

Время ответа довольно критично. Тот сервис «кушал» порядка 300 микросекунд (это с учетом TCP-запроса и ответа). Здесь мы проставили себе планку 200 микросекунд.

Сервис должен быть масштабируемым. Сама постановка задачи позволяет это сделать. Например, четные пользователи хранятся на одной машине, нечетные — на другой.

hl++ HighLoad++

Задача

- Периодически нужно делать бекап базы из памяти на диск
 - быстрый подъем при сбое
 - анализ данных по файлам без опроса
- Сервис должен быть легко расширяемым по функционалу
- сервис должен быть масштабируемым

Периодически нужно делать бэкап базы из памяти на диск. Почему? Во-первых, сбой самого сервера по каким-либо причинам. Во-вторых, связь. Иногда в Питере выключают свет. Делая бэкап, мы можем быстро подняться при сбое и восстановить хотя бы частично часовой snapshot памяти.

Вполне реально скопировать восстановленные данные на другую ненагруженную машину. Затем их можно анализировать без дополнительного опроса и без нагрузки на сервер. Он работает с продакшна, зачем его грузить лишний раз.

Сервис должен быть легко расширяемым по функционалу. Уже на момент переписывания было 3 UDP-клиента на обновление данных и два TCP-клиента на опрос данных. У каждого свой протокол. Пытались его немножко унифицировать. Посмотрим, что получилось.

hi⁺⁺

HighLoad++

Как можно хранить данные

• Memcached?

- Насколько гибка логика удаления? +
- Бэкап базы из памяти на диск? +
- Как обновлять по UDP (различные клиенты)? +/-
- Кто реализует дополнительную логику (ретаргетинг)? -

Как можно хранить данные? Сразу пришло в память «Memcached». Сам я не большой специалист, почти не использовал. Только какие-то базовые вещи слышал, немного использовал, задавал вопросы, пытался найти ответы в постах в интернете.

Насколько гибка логика удаления? Вроде бы может удалять старые объекты. Вроде бы может удалять объекты, когда память каких-то лимитов достигла. Этот пункт присутствует.

Бэкап базы из памяти на диск. По-моему, «Memcached» DB (Database) умеет. Но я могу ошибаться. Вроде такой пункт присутствует. Хотя все ругают, но есть.

Как обновляем по UDP? Тут я не нашел ничего внятного. Вроде бы умеет. Нам точно не сойдет, потому что не все наши клиенты, которые обновляют по UDP, знают полностью rkey value. Rkey user и ID им известны, а rkey value не всегда знают. Им предоставлена только часть данных о пользователе. Все равно не подходит.

Кто реализует дополнительную логику (ретаргетинг)? «Memcached» — это сервер общего назначения. У нас частный, это наши проблемы. В итоге, пришлось переписать.

hi⁺⁺

HighLoad++

В каком виде хранить?

- | | |
|--|--|
| <ul style="list-style-type: none"> • user1 <ul style="list-style-type: none"> – лог1 – лог2 • userN <ul style="list-style-type: none"> – логM – логM+1 | <ul style="list-style-type: none"> • user1 <ul style="list-style-type: none"> – struct1 – struct2 • userN <ul style="list-style-type: none"> – structM – structM+1 |
|--|--|

В каком виде мы будем хранить пользователя? Любое событие в системе — это запись в лог. В логе пишется кто, когда, где и какой баннер видел, кликал на него либо еще что-то делал. Можем по каждому пользователю хранить список этих самых событий. В итоге, мы всегда знаем все. Это довольно гибко, но занимает много места в памяти.

Давайте скомпонуем. Из текстового лога сделаем какие-то структуры. Получилось меньше. Давайте еще уберем лишние данные, которыми мы не пользуемся. Еще меньше. Скомпонуем два лога в один, например, показ и клик по одному баннеру засунем в одну структуру. Еще меньше.

Какой взять: по логам или по структурам?

hl⁺⁺ **HighLoad++**

В каком виде хранить?

<ul style="list-style-type: none"> • + гибкость • - много памяти • - есть доп. обработка <p>Доводы:</p> <ul style="list-style-type: none"> • требования меняются редко • бонус: <code>sizeof(struct1) = const</code> 	<ul style="list-style-type: none"> • - гибкость • + меньше памяти • + нет доп. обработки <ul style="list-style-type: none"> • нужна ↑ производительность
---	--

Первый дает гибкость. Например, мы можем задать несколько скоростей. Показывать не больше чем 5 раз за 10 минут и не чаще чем 8 раз за 20 минут или выставить две скорости.

Второй менее гибок. Первый «кушает» много памяти. Второй «кушает» памяти поменьше. У первого есть доп. обработка. Даже если мы будем хранить логи в бинарном виде, все равно мы должны по ним всем пройтись, пытаясь что-то найти. У второго варианта можно создать такую структуру, которую можно подсунуть в таргетинг и ничего больше не делать.

Чем мы руководствовались, когда принимали решение? Нужна высокая производительность. Требования меняются крайне редко. Сервисы переписываются. Как бонус получаем `sizeof(struct1) = const`. Можем какие-то аллокаторы сами написать.

hl⁺⁺ **HighLoad++**

Резюмируем требования

- Время ответа ($<= 200$ микросек)
- Всего храним $67 * 2$ млн юзеров, в день меняется 250 млн объектов их описывающих (~ 67 ГБ памяти)
- До 3000 з/сек на обновление данных и столько же на их выборку
- Бекап базы из памяти на диск

Резюме требований. 200 микросекунд. 67 миллионов на 2 машины — это мы немножко заходим вперед. 67 миллионов на одной машине. Всего машин две. Но это в номинале. Иногда одна в штатном или внештатном ремонте, поэтому вся нагрузка идет на одну машину.

В день меняется примерно 250 миллионов объектов. Памяти около 67 Гб. В пике до 3000 запросов в секунду. Должны уметь делать бекап базы из памяти на диск и использовать его, когда поднимаем сервер.



Архитектура

- Однопоточный
 - нет блокировок
- Неблокируемый ВВ
 - poll, epoll, kqueue
- Постоянные соединения
- UDP и TCP клиенты
- Свои аллокаторы (new/delete)
- Свои определения 'самый старый' объект
- Простая логика сохранения (fork)

Архитектура. Пускай он будет однопоточным, тогда не будет блокировок. Чтобы однопоточный сервер держал такую нагрузку, должен использоваться неблокируемый ввод-вывод (ВВ). Poll, epoll, kqueue — в зависимости от дефайнов.

Соединения будут постоянными. Тут можно схалтурить при установке соединения и в disconnect написать, например, не такой хороший код. По крайней мере, сам сервер никогда самостоятельно не обрывает connect.

Есть как UDP, так и DCP клиенты.

Можем написать свои собственные аллокаторы. Зачем? Задача первая — избавиться от фрагментации. При написании своих аллокаторов, своих операторов new/delete мы можем дать собственное определение разным типам объектов. Это тоже классно.

Простая логика сохранения. Делаем fork, пишем. Предок продолжает работать, обрабатывает соединение, потомок записывает данные на диски. Все замечательно. Правда, он дополнительно «кушает» память. С этим можно бороться.



Сетевой ВВ

- Все сокеты неблок:
 - fcntl(sock, F_SETFL, O_NONBLOCK);
- Неблокируемый ВВ (мультиплексирование)
 - poll, epoll, kqueue
- Постоянные соединения
- UDP и TCP клиенты

Вопрос из зала: (Неразборчиво, тихо).

Денис Бирюков: Треды нельзя. Иначе блокировки придется вводить.

Сервер-модуль состоит из двух частей. Это сетевой ВВ и логика. Сетевой ВВ полностью взят из предыдущего проекта. Его почти не переписывал, только какие-то аллокаторы добавил. Все сокеты неблокируемые. Функции мультиплексирования, постоянный ВВ, постоянное соединение, UDP, TCP клиенты.

hl++ HighLoad++

Протокол

```
struct MSG_TAB {
    int32_t size;
    int32_t func;
};

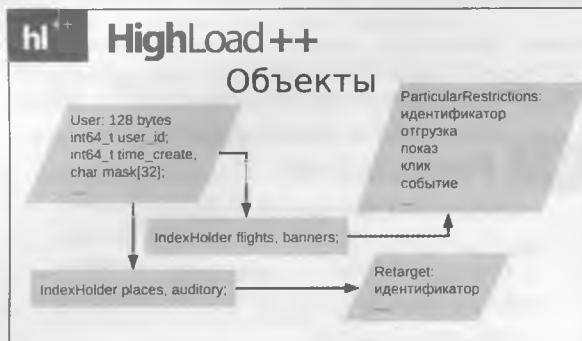
struct MSG_DATA {
    MSG_TAB h;
    char data[h.size];
};
```

- TCP, UDP:
 - read(...) - заголовок
 - readv(...) - данные
 - Action(...) - обработка
- TCP:
 - writev(...) - заголовок, данные

Как мы общаемся? Мы всегда работаем с бинарными данными. Сервер всегда в самом начале этих данных принимает и отправляет какой-то заголовок. Заголовок — это 8 байтов. Это размер данных и функция, которая должна обрабатывать эти данные.

По TCP запросу сервер сначала читает заголовок «8 байтов». Затем функция «readv», чтобы уменьшить нагрузку на сервер, читает сами данные, производит их обработку.

В случае с UDP мы не можем сделать «read» два раза, поэтому мы пытаемся при помощи «readv» прочитать столько, сколько можем — 8 байтов заголовка и 30 килобайт данных. Запросы от тех, кто не смог уложиться в этот объем, игнорируются. Данные пришли неполные, поэтому мы посылаем сообщение о том, что не успели их обработать. После обработки записываем заголовок и данные в результат, в поток вывода.



Создадим иерархию объектов, с которыми будем работать. Есть пользователь со своим идентификатором, временем создания, битовой маской, которые описывают тот самый `user_id`. У него есть контейнеры, которые хранят `flights`, баннеры. Они все одного объекта `ParticularRestrictions`. Указаны места, на которых он был, и аудитории, в которые он входит. Объекты типа `retarget`.

Почему контейнер `IndexHolder`? Вектор не подошел, потому что очень тяжело писать аллокатор для контейнера, который просит память то по два элемента, то по четыре, то по восемь. Короче, фрагментация.

У вектора в большинстве реализаций `capacity` в полтора раза больше, чем `size`. Это дополнительный расход памяти. Зависит от реализации, конечно.

Можно использовать лист. Фрагментации нет. `Capacity` равно `size`. Однако требуется дополнительная память под хранение данных под двунаправленный список. Нам это не надо.

`IndexHolder` — это массив, односторонний список, в котором выделяются массивы по 4 указателя. Первые три используются под наши нужды. Последний указывает на следующий массив из четырех указателей. В общем, такая у нас структура объектов.

hl⁺⁺ HighLoad++

Объекты, «устаревание»

- `User` — есть массив памяти под объекты
 - старый тот кто дольше всех не появлялся в сети (+ память на 2-х связный список)
- `IndexHolder (ArrayList)` — есть массив объектов
 - + 2-х связный список — есть избыток
- `ParticularRestrictions (Flight, Banner), Retarget (Place, Auditory)` — циклический массив.
 - самый старый по времени создания

Как мы теперь говорим, что объекты устарели? `User` — это массив памяти под объекты. Будем считать, что самый старый тот, кто дольше всех не проявлял активности в сети.

Мы дополнительно храним память на двухсвязный список из ID пользователей. Каждый раз, когда пользователь проявил какую-то активность в сети, мы его переносим из середины списка в начало. Соответственно, в конце будут самые старые по времени модификации.

`IndexHolder` — то же самое. Правда, потом получилось, что 2-х связный список для `IndexHolder` — это избыток. Первоначально планировалось обрабатывать exception через `bad_alloc`.

`ParticularRestriction` и `Retarget` — это обычные циклические массивы, большие куски памяти. Дошли до конца, идем сначала. В этом случае самый старый тот, кто был самым старым по времени создания.

hl⁺⁺ HighLoad++

Как храним user?

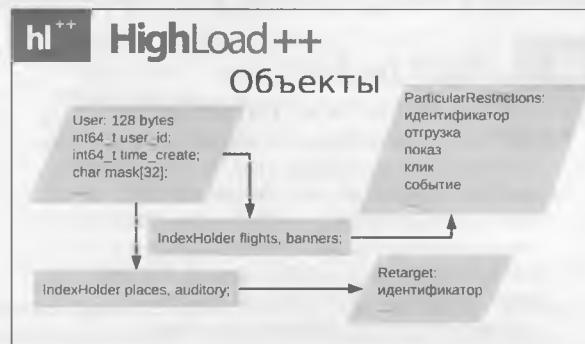
- `User` хранятся `std::map<u_int64_t, User*>`
 - `map` — не самый быстрый контейнер, но зато легко выделять память для него
 - для убыстрения работы создадим 64 `std::map`
- Память под `std::map` выделяется блоками по мере необходимости. Есть небольшой массив — хранит ссылки на свободные участки памяти

Как мы храним самих пользователей? Класс `user`. `Map` — конечно, не самый быстрый объект, но для него легко выделять память.

Добавили одного пользователя в `map`. Он говорит: «Выдели мне дополнительное место под один объект». Для `map` легко писать аллокатор. Чтобы избавиться от балансинга и ребалансинга, создадим 64 `map`. Хотя у нас `user ID` — это довольно рандомная величина.

Память под `map` выделяется не одним большим куском, а блоками, по мере необходимости. Есть какой-то небольшой массив-указатель, в котором хранится указатель на свободные участки памяти. Как только этот массив кончается, мы опять выделяем блок памяти и заполняем этот массив.

Если мы удалили user'а из map, мы засовываем указатель на память, где был этот user, в этот map, и используем его заново.



Как мы удаляем объект? Мы взяли самого старого user'а по времени модификации, удалили все объекты ParticularRestriction и Retarget. Вернее, пометили память, которую они занимают, как свободную. В следующий раз, когда будем аллоцировать, все будет замечательно. На этом месте опять создаем нового user'а.

Что будет, если мы хотим создать объект ParticularRestriction? Берем очередную порцию в циклическом массиве, кусок памяти, и смотрим, свободный ли он. Если свободный — все замечательно. Если нет, то мы нотифицируем. Здесь содержится указатель на конкретный экземпляр класса user. Мы нотифицируем user'а о том, что он должен почистить в своих контейнерах flight или баннер и убить этот указатель.

ParticularRestriction содержит parent ID. Parent ID у flight равен минус 1, у баннера parent ID какой-то вменяемый. Четко, ясно, где искать: во flight или в баннерах. Аналогично с ретаргетингом.



hl⁺⁺

HighLoad++

Сохранение на диск

- По специальной UDP датаграмме делаем fork
- Дочерний процесс читает и записывает на диск 4 куска памяти: User, ParticularRestrictions, Retarget, IndexHolder. Целостность данных обеспечена.
- Результаты $(0,75+4+1)/(9,375) \sim 61\% \text{ КПД} (> \text{в 1,6 раза})$:
 - User — 2,125 Gb (1,5 Gb (main) + 0,625 Gb (add))
 - IndexHolder — 1,5 Gb (1 GB (main) + 0,5 Gb (add))
 - ParticularRestrictions — 4 Gb
 - Retarget — 1 Gb
 - std::map — 0,75 Gb

Сохранение на диск теперь довольно простая операция. Делаем UDP пакет. Посыпаем UDP пакет на сервер, он делает fork. Предок продолжает работать, меняет страницы памяти. Продок записывает 4 куска на диск.

Результаты. 1,5 Гб под user'a, дополнительно 0,6 Гб под 2-х связный список. Аналогично для IndexHolder, ParticularRestriction, Retarget. Под std::map максимум 0,75 Гб.

Почему так? Если убрать отсюда все лишние данные и сравнить, что в памяти и что в очень сжатом виде, то получим КПД 61%. Лучше, чем в 3,5 раза. Это уже хорошо.

Почему цифры именно такие? На самом деле на машине запущено 4 экземпляра сервера, поэтому эти цифры нужно смело умножать на 4 для одной машины.

hl⁺⁺

HighLoad++

ЕМКОСТЬ ПАМЯТИ

- По прикидкам скорость просмотра: 1 ~ 1,5 баннера / 1 мес.
- Время жизни всех объектов ~ 1 мес.
- Потеря активного flight / banner, менее критична чем потеря профиля пользователя.
- Если потеряем старые auditory (place), то охват уменьшится, но качество повысится.
 - User — 67 108 864; std::map — сколько нужно
 - ParticularRestrictions — 134 216 960
 - Retarget — 134 217 728
 - IndexHolder — 268 435 456 > 134 216 960 + 134 217 728 !

Давайте прикидывать, какие константы пропишем для аллокаторов. Емкость просмотра у нас не очень высока — от 1 до 1,5 баннеров в месяц на user'a. Время жизни всех объектов приблизительно месяц.

Потери активного flight/баннера для нас менее критична, чем потеря пользователя. Рекламная кампания закончилась. Flight/баннер не нужен, а пользователь еще может посмотреть новую рекламную кампанию. Очень не хочется терять profile этого пользователя, доставшегося нам такой дорогой ценой. Flight/баннер у нас имеет циклический массив, а user удаляется по времени модификации.

Если потеряем старые auditory/place (они нужны для ретаргетинга), то охват мы, конечно, уменьшим, но качество повысим. Мы не будем показывать тем, кто давно интересовался какой-то темой. Покажем тем, кто недавно проявил заинтересованность.

Это цифры для одной машины. Почему я говорю, что IndexHolder избыточен? Если IndexHolder у нас больше, чем количество объектов ParticularRestriction и Retarget, то никакого bad_alloc'a не возникнет только в случае, если у нас утекает память под IndexHolder.

Логика. Action().

```

switch(readheader.func) {
    - case (SET_USER_EVT): юзер сделал хит
    - case (SET_USER_MASK): юзер изменил профиль
    - case (SET_USER_ADT): юзер добавлен в аудиторию
    - case (SET_RETARGETS): обновили ретаргетинги
    - case (SET_CAMPAIGN): обновили уникальность по РК
    - case (GET_USER_EXP): описание юзера, выдаем банер
    - case (GET_USER_EVT): описание юзера, проверяем клик
    - case (UU_CONTROL): служ: fork, fork_info, mem_info
}

```

Теперь логика. Там был сетевой вывод, а тут — логика. Объект Action у нас имеет заголовок и данные. Switch по номеру функции и конкретному событию (case). User сделал какой-то хит, мы его пытаемся найти. Если его нет, то создаем, добавляем или модифицируем объекты flight, banner или place в нем.

Case SET_USER_MASK — мы изменили profile user'а и пытаемся его найти. Если нашли, то поменяем его битовую маску.

Case SET_USER_ADT — добавляем пользователя в какую-то аудиторию. Аудитория у нас поисковая. Какой-то онлайн-сервер постоянно ищет. Пользователь искал в Google или в Яндексе какое-то интересное слово. Если нам это слово или фраза интересны, то мы его добавляем. Затем обновляем ретаргетинги.

Что такое ретаргетинги? Вкратце было видно. Есть пользователи. Они хранят информацию о том, где и что они видели. Есть текущие настройки ретаргетинга. Например, мы захотели показывать flight № 10 тем, кто был на place № 8. Накладываем одно на другое и добавляем в список ретаргетингов.

Список ретаргетингов. Мы запрашиваем в GET_USER_EXP описание юзера, чтобы рассчитать, какой баннер показать. Здесь мы показываем все flight, все баннеры. В конце находятся идентификаторы flight, ретаргетингов, которые можно показывать этому пользователю. Туда добавляем эти ретаргетинговые flight'ы.

SET_CAMPAIGN — это новая фича. У flight внутри есть список баннеров. Flight могут быть объединены в список компаний. Не хотелось создавать отдельный объект, поэтому сделали как в ретаргетинге. Flight № 1 может показываться 2 раза. Flight № 3 может показываться 5 раз. Тогда мы можем сказать, что flight № 1 и flight № 3 показывать можно. Если нет, то не добавляем их в список, лишаем их показа.

«SET_USER_EVT» — например, пользователь сделает клик. Как мы зачтем или не зачтем этот клик? Если он видел баннер, то есть ему баннер отгрузили, то, наверное, зачесть можно. Если он не видел, то клик не засчитывается, хотя redirect отдается.

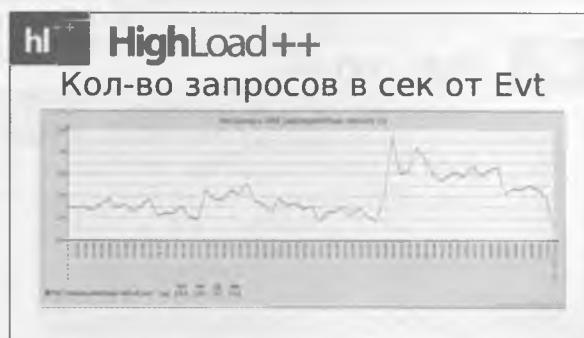
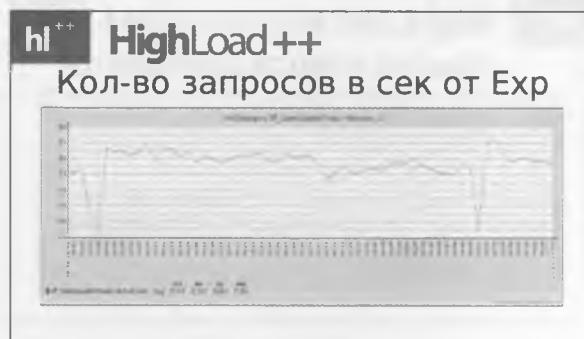
Антиклик. Нельзя засчитывать клик, если пользователь кликает чаще одного раза в пять минут.

UU_CONTROL — это контроль и управление сервером. Специальным пакетом делается fork. Потом fork_info выдает информацию о том, какой pid дочернего процесса был, в какое время создан текущей машиной. Напомню, у нас 4 экземпляра на одном сервере. Если все вместе будут делать fork, то нам нужно еще столько же памяти. Мы лимиты свои исчерпаем, поэтому их 4 экземпляра, а не один.

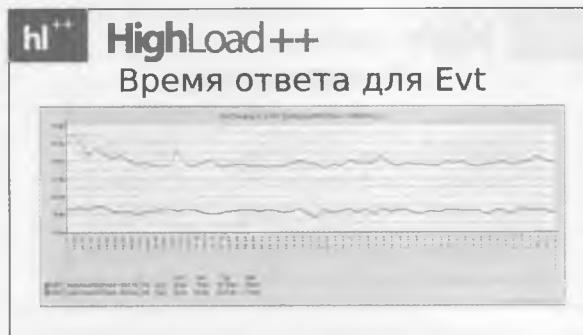
Mem_info показывает, сколько памяти выделил текущий экземпляр у сервера, сколько осталось. Память, в принципе, выделена, часть из нее помечена как свободная.



Прежде чем показать данные по нагрузке... Отдельный квадрат — это машины с экземплярами. Front — это основные потребители данных от этих машин. Каждый с каждым соединяется. Когда смотрим график, там видна вся информация по одной машине.



Теперь количество запросов в секунду от event'a. Оно гораздо меньше. Примерно 2500.



Время на 20 микросекунд меньше. Среднее 140 микросекунд.



Обработка внутри самого сервера занимает 7 микросекунд.



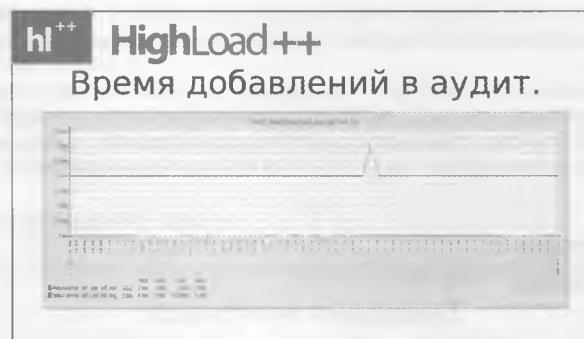
Сколько мы отправили UDP пакетов на обновление? Если сравним эту цифру (в среднем 38,3 тысячи), то предыдущие данные должны плюсоваться 36,5 и два или три. Эти цифры должны быть равны. Значит, с сервером у нас все нормально. 38,3 — все нормально.



Время обработки одного UDP пакета в районе 5 микросекунд.



Сколько user'ов добавляется в аудиторию? В среднем 4000 в минуту. Иногда бывают всплески. Мы анализируем Sphinx'ом статистику. Вдруг мы захотим добавить новую аудиторию, и у нас уже будут какие-то пользователи по ней.



Время добавления в аудиторию в районе 4-х микросекунд.

hi++ HighLoad++

Резюме по боевой нагрузке

- Кол-во TCP запросов: 37 тыс + 2,6 тыс ~ 39,6 тыс з/мин
- Кол-во UDP пакетов: 38,3 тыс з/мин
- Кол-во аудиторных UDP пакетов: 4 тыс з/мин.

Итого на сервер сейчас в номинальном режиме:

- ~ TCP 670 з/сек.
- ~ UDP 680 з/сек.

А каков предел по нагрузке?

Резюме по боевой нагрузке. У нас TCP запросов в номинальном режиме 670 в секунду. Плюс UDP столько же.

Каков предел по нагрузке? Сделали специальную тестовую утилиту. Сначала она посыпает UDP пакет на создание user'a, потом отправляет TCP-запрос и ожидает TCP-ответ.



Если сделать тест, запустить 50 потоков, которые опрашивают одну машину из четырех, то полмиллиона таких запросов обрабатывается за 20 секунд. Дальше нагрузка линейная, вполне ожидаемая.

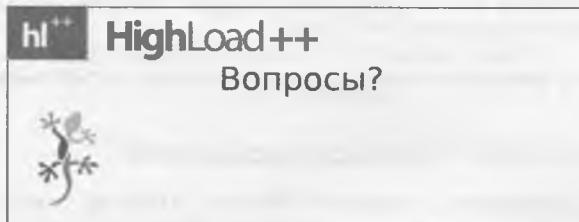
Если мы запустим больше потоков (100 потоков), то они будут это все гораздо быстрее делать, за 10 секунд. Почему? Видимо, сервер справляется с нагрузкой, а клиенты имеют какое-то строго фиксированное время одного запроса. Чем больше одновременно запущенных клиентов, тем быстрее.



Как это переварить? На графике представлено количество запросов в секунду. Если сделаем 50 клиентов, то количество запросов будет в районе 25-ти тысяч. Если 100 клиентов, то 55 тысяч. Самый пик — это 180 потоков — будет в районе 85-ти тысяч.

Тестовая машина AMD Opteron. Два ядра по 8 процессоров в каждом. Или два 8ми ядерных процессора? Короче, 8 ядер.

Все, спасибо.



Вопросы и Ответы

Вопрос из зала: Вы сохраняете пользователей. По каким критериям вы определяете, кто есть кто? Сам идентификатор этого пользователя.

Денис Бирюков: Сначала мы даем ему cookies. Если он второй раз пришел с этими cookies — замечательно, то есть cookies ему проставились. Если нет, то мы ему заново генерируем cookies или считаем, что он плохой.

В принципе, наш охват — 75 миллионов — это, конечно, нереальный охват. Реально 20 миллионов. Сколько cookies мы генерировали для 75-ти миллионов? Это большая проблема. Синхронизация cookies, в том числе.

Мы думали ввести сложный алгоритм на основании IP адреса, user-агента, еще чего-нибудь в этом духе, но пока не ввели. Не очень понятно, какова ситуация будет.

Сейчас просто генерируем cookies на основании текущего timestamp, времени, IP-адреса и user-агента. Если одновременно к нам пришло 3 запроса от страницы, выполненных примерно в одно и то же время, то мы генерируем один cookie. Если нет, то нет. Последний будет потерян.

Вопрос из зала: (Неразборчиво, тихо).

Денис Бирюков: Этот сервер статистику не считает. Вообще статистика считается Hadoop'ом отдельно по логам.

Вопрос из зала: Если у вас свой аллокатор, то зачем вы используете voit*, а не просто индекс?

Денис Бирюков: На самом деле там не voit*. Там int index в глобальном массиве.

По поводу статистики. Этот сервер нужен для того, чтобы быстро отдать баннер. Время ответа 200 микросекунд. Иначе мы баннер быстро отдадим.

Мы храним все в логах. Логов у нас несколько терабайт за несколько дней, точные цифры я вам не скажу. Короче, Hadoop как-то все разгребает.

Вопрос из зала: Почему в качестве контейнера использовался map, а не hash map?

Денис Бирюков: Мы хотим добавить одного user'a. Map требует ровно 48 байт информации по нему. Hash map, как и вектор, может потребовать больше, чем ему полагается. Тяжело описать такой аллокатор, фрагментации будет много.

Вопрос из зала: (Неразборчиво, тихо).

Денис Бирюков: С кликами мы боремся. У нас нельзя кликнуть по баннеру, если он тебе предварительно не отгружен. Мы ссылку отдадим, user подумает, что он реально кликнул. Однако в статистике мы не сохраним этот клик.

Если он за одну минуту два раза кликнет, то мы посчитаем только первый раз. Второй раз считать не будем.

Вопрос из зала: Борьба с кликами идет в реальном времени?

Денис Бирюков: Да, борьба идет в реальном времени. В Hadoop тоже можно какие-то алгоритмы вводить, но пока мы еще не вводили. Все пока в реальном.

Вопрос из зала: Алгоритмы по изменению в логах или баннерах не используются?

Денис Бирюков: Нет. User'у отгрузили баннер, он может по нему кликнуть. Там есть сложный отдельный сервис. Я про сетевой вывод говорил. Это отдельный сервер от накрутки. Мы будем пытаться прекращать отгрузку баннеров всяким cheat'ерам.

Допустим, нас долбанули каким-нибудь benchmark'ом. Мы будем давать ответы, но при этом не грузить основной функционал сервера.

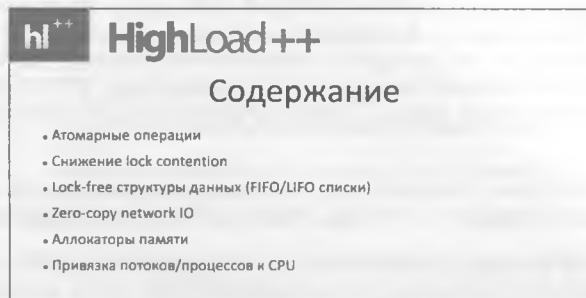
Спасибо.

Как построить высокопроизводительный front-end сервер

Александр Крижановский



Александр Крижановский: Доброе утро! Сегодня я расскажу про front-end сервер.



Буду говорить даже не столько про сам front-end сервер, сколько про некоторые паттерны (шаблонные) его реализации. Мы рассмотрим некоторые задачи, которые часто встречаются при разработке высокопроизводительных серверов, и познакомимся с их решениями.

В основном сегодня будем рассматривать очереди. Как делать быстрые очереди, разделяемые между одним producer'ом и несколькими consumer'ами. Zero-copy ввод-вывод и атомарные операции. Больше поговорим о concurrency.

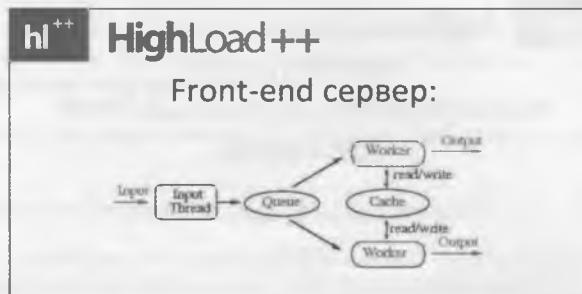


Сначала определимся, что имеется в виду под front-end сервером. Кто-то его знает как Nginx, который стоит перед Apache, кто-то его знает как сам Apache, который стоит перед MySQL.

Когда я делал эту презентацию, я думал о том, как лучше его назвать. Здесь имеется в виду обычный сервер — DNS-сервер, веб-сервер, — который обращается только к первичной па-

мяти (RAM). Он редко или никогда не обращается к диску. Таким серверам свойственно иметь высокие требования к внутренним вычислительным операциям. Редкие обращения к диску дают высокую производительность.

Такие сервера обычно являются многопоточными. Между собой эти потоки что-то делают. Обычно это какой-то producer и consumer'ы, которые разделяют общую структуру данных.

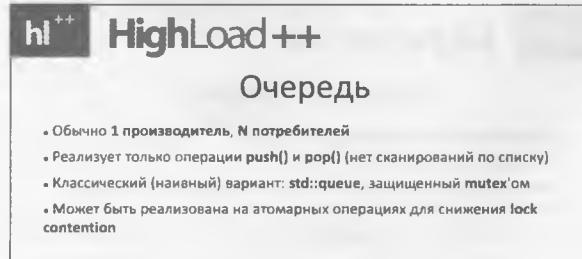


На слайде представлена архитектура, о которой мы сегодня будем говорить. Один поток ввода. Часто это поток, который крутится на epoll либо select (системные вызовы мультиплексирования). Он через очередь поставляет либо socket'ы, либо уже считанные данные из всех socket'ов на нескольких worker'ов. Если это http-сервер, worker'ы отрабатывают сам socket, принимают http-запрос, смотрят в различные кэши (HTTP соединений, ответов и пр.), формируют и отправляют ответы. В этом случае каждый worker сам отдает ответ на socket'ы.

В очередь пишут одни потоки, а читают другие. В кэш часто и пишут и читают из него одни и те же потоки. В обоих случаях — это очень 'горячие' структуры данных.

Есть архитектура, когда worker'ы сами ничего не отправляют в socket'ы, а просто передают данные другому потоку, который на слайде не показан. Этот поток выполняет вывод. Имеем поток ввода и много быстрых worker'ов, которые делают операции, парсят данные, собирают их, преобразуют и отдают дальше во вторую очередь на вывод.

Самая частая архитектура — это Xeon'ы с 8-ю или 16-ю ядрами, подключенные к гигабитному каналу. Современное ядро Xeon 2,8 — 3 ГГц вполне может обрабатывать 1 Гбит трафика на ввод и/или вывод. Архитектура с одним потоком, привязанным к одному ядру и выполняющим все операции ввода-вывода, живет обычно успешно. Это зависит от ваших данных, объемов запросов и того, как часто происходит копирование между user space и ядром.



Сначала рассмотрим очередь. Очередь — это какая-либо структура данных, реализованная либо на массиве, либо на списке. Для нашего случая характерно, что у нас всего один производитель и куча потребителей.

На этой структуре данных определены только две операции — `push()` и `pop()`. Здесь нет сканирования, сортировки и прочего. Всего две операции, которые оперируют с двумя полями этой структуры данных — головой и хвостом.

Мы сегодня в равной степени будем говорить и об очереди, и о стеке. Для нас они сегодня будут эквивалентными.

Самая простая реализация — это обычная С-шная очередь на указателях или массиве (*ring buffer*) либо `std::queue` (для C++). Все это защищается `mutex`'ом. Мы здесь можем использовать только взаимоисключающую блокировку, потому что это очередь как на запись, так и на чтение. Обычно очередь не получается хорошо защитить блокировками.

Такое решение хорошо живет, пока у вас мало процессов, то есть пока нет высокой «конкуренции» между *worker*'ами за право считать из этой очереди. Самая быстрая реализация, которая часто используется, — это *lock-free* очереди. Эти методы могут быть применены только к очередям, к таким структурным данным, где определены только эти две операции — `push()` и `pop()`.



На всякий случай поговорим немного про *Lock contention*. Его не всегда легко увидеть. У вас есть сервер, он работает. Вы открываете `top` и видите, что CPU не используется, по памяти все хорошо, `swapping`'а нет, `load average` низкий. Система говорит, что у нее все хорошо, нагрузка небольшая.

Вы запускаете какой-нибудь *benchmark* и смотрите, что он крутится, но при этом пропускной способности системы не хватает. Это условие *Lock contention*. Возникает он из-за того, что сервер резко становится однопоточным. Всего один поток удерживает блокировку на очереди. Пока этот процессор пытается считать или записать какой-то указатель в очередь, другие процессоры простояивают.

Это одна из самых дрянных проблем в производительности. С ней тяжело бороться. Чтобы ее побороть, обычно нужно очень много всего сделать. Часто, чтобы побороть *Lock contention*, изменяют архитектуру всего проекта.

Если у машины всего два-четыре процессора, то это ерунда. Здесь *Lock contention* небольшой, все процессы обычно успевают, потому что они не все синхронно «ломятся» в очередь. Сейчас есть машины с 32-мя ядрами. Иногда количество запущенных потоков превышает количество ядер. Получается, что 32 потока могут ломиться на одну очередь. Имеем большой *Lock contention*, дорогое железо фактически простояивает.

Первый метод блокировки — использование безлоковых операций. Безлоковыми они являются для программистов. В реальности это не совсем так. Однако это самое простое, что мы можем сделать. Второй метод, который хорош для борьбы с более сложными структурами данных, — увеличение гранулярности блокировок.

hl ++

HighLoad++

Блокировки

- `pthread_mutex`, `pthread_rwlock` - используют `futex(2)`, достаточно тяжеловесны сами по себе
- `pthread_spinlock` («busy loop») — в ядре ОС используется в сочетании с запретом вытеснения, в user space может привести к нежелательным последствиям
- Атомарные операции, барьеры и `double-check locking` — наиболее легкие, но все равно не «бесплатны»

Прежде всего, о том, какие блокировки доступны в Linux. Это хорошо известный `pthread_mutex`. Это взаимная блокировка, которая реализована по принципу семафора — либо вы ее захватили, либо нет. Если вы захватили, вы можете читать и писать в структуру. Если нет, тогда пристаиваете в ожидании.

Эта блокировка сама по себе довольно тяжеловесная. Если вы работаете на однопроцессорной машине, то все более или менее нормально. Там достаточно много кода, но он быстро проскачивает.

С многопроцессорными системами сложнее. Если вы начинаете блокироваться, то при каждом запросе на блокировку, когда вам нужно подождать, вы на самом деле уходите в ядро. Делается это через системный вызов `futex(2)`. `Futex(2)` позволяет поместить вызвавший поток в очередь ожидания, чтобы он заснул и не использовал процессор. Когда блокировка становится доступной по конкретному адресу в памяти, то ядро пробуждает все потоки, которые находятся в ожидании. Потоки захватывают блокировки.

Из ядра в user space перешли `spinlock`. `Spin` — это волчок. `Spinlock` — это цикл ожидания. Грубо говоря, если вы напишите цикл `while`, который ожидает на переменной определенного значения, то это будет примерно `spinlock`. Один процесс присваивает переменной значение 1, заходит в критическую секцию. Другие процессы крутятся в бесконечном цикле до тех пор, пока переменной не присвоят значение 0. Кто первый успеет, тот снова присвоит ей значение 1.

В ядре это работает. Когда поток захватывает `spinlock`, он запрещает `preemption`. `Preemption` — это механизм, через который ядро может вытеснить задачи. Если поток в ядре захватил `spinlock`, то это поток больше не перепланируется. Он гарантировано закончит за отведенный квант времени все операции по `spinlock`'ам.

В user space это не так. Мы не можем сказать, что нельзя вытеснить наш поток. Может получиться неприятная ситуация.

Один поток захватил `spinlock`, выполняет операции под `spinlock`'ам, перепланируется. Его вытесняют другие потоки. Другой поток пытается захватить `spinlock`, крутится на нем, так как тот недоступен, и выжирает полностью процессор. То есть выжирает весь свой квант времени. Другие потоки ведут себя так же. В этот момент мы увидим, что все ядра полностью используются процессором.

Потоки перепланируются при исчерпании кванта времени. Ситуация уже нехорошая. Ситуация становится хуже, когда планировщик разделяет real-time задачи и числовые дробилки. Пример real-time процесса — графический редактор. Мы подумали некоторое время, потом выполняем какое-то действие. Операционная система должна отвечать быстро, но процессора при этом мы потребляем мало.

Числовые дробилки — это агрессивные задачи, которые выжирают все доступное им время. Они всегда имеют меньший по сравнению с real-time задачами приоритет.

У задач, которые хотели попасть в этот spinlock и выжрали свой квант времени, понижается приоритет. В следующий раз получить нужную блокировку им будет еще тяжелее. Поэтому spinlock в user space использовать не надо.

Первые две блокировки мы рассматривать не будем. Мы сегодня будем больше говорить про атомарные операции. Про double-check locking и барьеры я скажу совсем немного.

Думаю, что все знают ключевое слово языка C — volatile. Оно говорит компилятору о том, что не нужно делать никаких предположений о хранимом значении в переменной. Не нужно его кэшировать в регистры. Это значение нужно всегда брать напрямую из памяти, потому что оно может в любой момент измениться.

Volatile — это уровень компилятора. Memory-барьер — это уровень процессора. Однако механизм похож. Если вы устанавливаете в коде барьер памяти, то все операции до этого барьера должны завершиться. Тип операций — на чтение, на запись, все операции — зависит от типа барьера.

Это некоторый барьер на конвейере процессора, через который нельзя перемешивать инструкции. В обычном случае процессор перемешивает все инструкции чтения-записи, отслеживает зависимости по данным между этими инструкциями. Так он ускоряет работу с памятью и полностью использует свой конвейер. Когда мы выставляем memory-барьер, мы говорим процессору, что до этого момента все должно завершиться и инструкции не должны приходить через этот барьер.

Double-check locking (второе название — lazy lock). Нам нужно взять блокировку на запись на какую-то структуру данных. Сначала мы берем блокировку на чтение, а затем смотрим, нужна ли нам блокировка на запись. Если такая потребность есть, мы ее берем.

Приведу пример. У нас есть кэш, который мы проверяем на наличие некоторого элемента. Если элемент отсутствует, мы вставляем его. Первая операция — блокировка на чтение кэша. Проверяем наличие элемента. Если элемент есть, мы его берем.

Если элемента нет, то мы отпускаем блокировку на чтение и снова берем блокировку, но уже на запись, и снова проверяем наличие элемента. Зачем? Между тем, как мы отпустили блокировку на чтение и взяли блокировку на запись, кто-то может вставить элемент. Мы должны снова повторить поиск по этой структуре данных. Если в этот раз мы не находим элемент, то тогда вставляем его.

С одной стороны, у нас два прохода по структуре данных, выполняется больше работы. С другой стороны, у нас снижается Lock contention (т.к. блокировку на чтение могут удерживать несколько потоков, а на запись — только один). Мы либо боремся за общую производительность на одном ядре, либо позволяем большему числу процессов выполнять одновременно.

hl ++ HighLoad++ Атомарные операции (Read-Modify-Write)

```
unsigned long a, b = 1;  
b = __sync_fetch_and_add(&a, 1);  
  
mov $0x1,%edx  
lock xadd %rdx,(%rax)
```

Все операции чтения-записи на современных процессорах (x86-64 и популярные RISC-процессоры такие, как SPARC и Power) атомарные. Если мы читаем или пишем в целое число, то эта инструкция гарантировано атомарная.

Но допустим мы хотим сделать инкремент переменной. В данном примере мы хотим увеличить переменную *a* на единицу и присвоить результат переменной *b*. Эта операция называется Read-Modify-Write. Мы сначала читаем значение переменной из памяти, увеличиваем его на единицу и кладем снова в память. Если два процессора выполняют эту операцию параллельно, они могут считать одно и то же значение, оба его инкрементировать и записать обратно. У нас получается один инкремент вместо двух.

GCC предоставляет встроенные функции для атомарных операций. Сейчас не нужно писать эти вещи на assembler. Мы можем использовать `_sync_fetch_and_add(&a, 1)` и другие функции (вычитание, битовые операции и т.д.)

Хочу обратить внимание. Эти функции гарантируют атомарный инкремент для *a*. Они не гарантируют, что вся операция будет атомарной — чтение *a*, его увеличение и присвоение значения *b*. На слайде представлен листинг кода assembler, в который разворачивается данная встроенная функция. Атомарность заключается в добавлении префикса `lock` к некоторым инструкциям.

Все это работает на уровне процессора, на уровне контроллера памяти. Процессор выполняет инструкцию с префиксом `lock`. При этом он гарантирует, что другие процессоры, которые хотят выполнить инструкцию на ту же область памяти, будут стоять в ожидании.

Наша атомарная операция — это `xadd`. Она добавляет единицу к переменной *a* и возвращает результат. Затем будет инструкция `mov`, которая будет перемещать значения из регистра *rax* в регистр, который соответствует переменной *b*.

Операция `lock` связана с управлением памятью и со взаимодействием с другими процессорами, которые находятся в системе. Это означает, что операция `lock` достаточно медленная. Часто после нее процесс перепланируется. На практике очень часто встречается ситуация, когда переменная *a* уже инкрементирована, процесс перепланируется, но переменная *b* не имеет значения. Присвоение переменной *b* не атомарно.

The screenshot shows the HighLoad++ IDE interface. At the top, there's a toolbar with icons for file operations and a search bar. Below that is a header bar with the text "hl++" and "HighLoad++". The main area contains a code editor and a assembly dump window. The code editor has the following content:

```
unsigned long val = 5;
__sync_val_compare_and_swap(&val, 5, 2);

    mov $0x5,%eax
    mov $0x2,%ecx
    lock cmpxchg %rcx,[%rdx]
```

The assembly dump window below shows the generated assembly code:

```
    mov    $0x5,%eax
    mov    $0x2,%ecx
    lock  cmpxchg %rcx,[%rdx]
```

Предыдущая инструкция хорошо подходит для статистики, когда нам нужно увеличить некоторые counter'ы или реализовать простейшую блокировку. Сегодня мы больше будем смотреть на инструкцию Compare-And-Swap (CAS). В данном примере мы сравниваем значение, которое находится в переменной *val*, с ее старым значением (5). Если оно не изменилось, тогда мы присваиваем новое значение 2.

Инструкция очень полезна для работы с очередями для того, чтобы мы могли атомарно переставить указатели. При помощи этой инструкции мы можем быть уверены, что никто не добавит новый элемент. Это позволяет безопасным образом подменить указатели.

`Cmpxchg` нуждается в префиксе `lock`. Инструкция `xchg` просто меняет два элемента местами (`swap`) и не нуждается в префиксе `lock`, поскольку всегда атомарна. Самый лучший ме-

тод поменять два значения местами — использовать инструкцию `xchq`. При этом блокировки не нужны. Кроме того, не нужно выполнять классические три инструкции — присвоить во временное хранение, временную переменную присвоить первой и так далее.

hi⁺⁺ HighLoad++

Атомарные операции: стоимость

- Реализуются через протокол **cache coherency** (MESI)
- Писать в одну область памяти на разных процессорах дорого, т.к. процессоры должны обмениваться сообщениями RFO (Request For Ownership)

Используются в `shared_ptr` для *reference counting*

При атомарных операциях в коде у нас никогда не бывает конструкций типа `pthread_mutex` и прочих вещей. Однако атомарные операции при этом дорогостоящие. Прежде всего из-за того, что они реализуются через протокол поддержки консистентности кэшей.

Рассмотрим пример. В системе два процессора работают с разделяемой памятью. В какой-то момент процессор 1 пишет в переменную `a` какое-то значение. В следующий момент процессор 2 пишет в эту же переменную другое значение. Поскольку процессор работает с кэшем, все значения кэшированы. Два кэша на разных процессорах соответствуют одной области в памяти.

Мы пытаемся записать в переменную значение. В хорошем случае у нас на процессоре 1 переменная уже закэширована, этот процессор полностью владеет правами на изменение данной области памяти. Когда процессор 2 пытается записать переменную, он не видит, что она уже есть в кэше. Он запрашивает это значение из памяти. На процессоре 1 происходит сброс кэша в память, затем — извлечение данных из памяти в процессор 2.

Когда мы работаем с разделяемыми данными, между процессорами проскачивает большое количество внутренних сообщений. Они призваны гарантировать консистентность кэша между процессорами. Процессоры при этом вовремя сбрасывают свои данные из кэшей в память. Кроме того, это гарантирует, что только один процессор в каждый момент времени может писать по конкретному адресу в памяти.

Такой способ на деле очень медленный. Возьмем два потока, которые будут писать в одну и ту же область памяти. Они даже на двухпроцессорной машине будут заметно медленнее, чем поток с удвоенной скоростью.

Все эти операции стоят прилично.

Есть несколько протоколов поддержки консистентности кэшей для разных архитектур. На Intel-архитектуре это протокол MESI. Сообщения, которые гарантируют, что вы изменяете только одну область памяти на одном процессоре, называются RFO.

hl++ HighLoad++

Lock-free очередь (список)

```
push (ff: pointer to fifo, cl: pointer to cell):
Loop:
    cl->next = ff->tail
    if CAS (&ff->tail, cl->next, cl):
        break
```

Рассмотрим простейшую динамическую структуру данных. Это очередь или стек.

Первая операция — это `push()`. Мы берем текущую голову списка, сохраняем указатель и пытаемся с помощью операции CAS изменить текущее значение головы списка на новый элемент `cl`. Подменяя его только в том случае, если оно еще не изменилось.

Для этого между первой операцией, когда мы присвоили `next` текущее значение головы списка, и следующей, никто не должен вставлять элементы в список кроме текущего потока. Если CAS завершается ошибочно, то есть новое значение не соответствует нашим ожиданиям, то функция уходит в следующую итерацию цикла. Это происходит до тех пор, пока не удовлетворится требование неизменности указателя на голову списка.

hl++ HighLoad++

Lock-free очередь (список)

```
pop (ff: pointer to fifo):
Loop:
    cl = ff->head
    next = cl->next
    if CAS (&ff->head, cl, next):
        break
    return cl
```

Пока все хорошо.

Следующая операция аналогична. Это операция `pop()`. Она чуть сложнее, потому что мы должны удерживать указатель на два элемента — на голову и на следующий элемент. Мы извлекаем текущий элемент, при этом голова списка должна указывать на следующий за ним.

Здесь тоже все хорошо. Мы убеждаемся в том, что никто нас не перепланирует, что список находится в консистентном состоянии. Это необходимо сделать перед тем, как мы попытаемся записать в значение головы списка элемент `cl`, который следует за текущим.

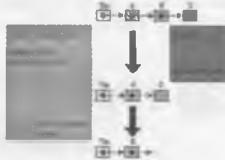
Хочу обратить внимание, что присвоение `next` находится не в самом CAS'е. Это не принципиально. Мы могли бы вместо переменной `next` напрямую использовать `cl` `next`. В реальности перед вызовом инструкции мы сначала обратимся в память и пройдем ссылку между `cl` и его элементом. Только после этого мы выполним саму инструкцию.

hl⁺⁺

HighLoad++

ABA problem

- Поток T1 читает значение A,
- T1 вытесняется, позволяя выполнятся T2,
- T2 меняет значение A на B и обратно на A,
- T1 возобновляет работу, видит, что значение не изменилось, и продолжает...



На самом деле, даже при том, что мы использовали атомарную инструкцию, здесь не все хорошо. Есть проблема АВА. Она соответствует ситуации, когда мы сохраняем предыдущее значение головы списка. При этом мы рассчитываем на то, что операция успешно завершится при условии, что никто не изменит значение головы списка.

hl⁺⁺

HighLoad++

Lock-free очередь: ABA

```
pop (ff: pointer to fifo):
Loop:
    cl = ff->head
    next = cl->next
    ----->scheduled
    if CAS (&ff->head, cl, next): # cl = A, next = B
        # pop(A), pop(B), push(A)
        break
    return cl
```

Перед тем как мы вызвали инструкцию CAS, мы перепланируемся. Какой-то другой поток сначала извлекает первый элемент, который мы хотим вытолкнуть, потом следующий за ним. В конце он снова добавляет первый элемент. Указатель на голову списка не изменится, а указатель на следующий элемент изменится. Когда мы подменяем указатели, у нас CAS пойдет без ошибки, но при этом получится битая ссылка внутри списка.

hl⁺⁺

HighLoad++

Решение АВА

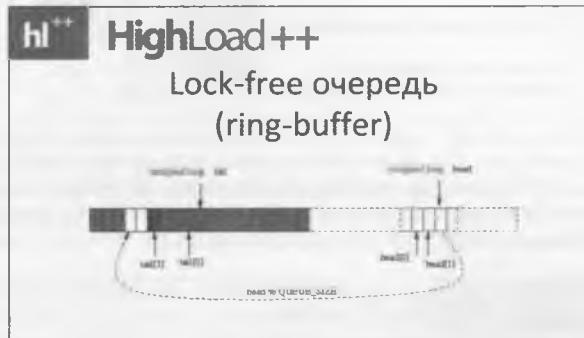
- Вводятся счетчики числа pop()'ов и push()'ей для всей структуры или отдельных элементов и атомарно сравниваются
- Нужна операция CAS2 (Double CAS) для сравнения двух operandов: **CMPXCHG16B** на x86-64

Эта проблема достаточно тяжело решается. Раньше, когда процессоры были 32-битными, у нас не было операции Double CAS. Эта операция позволяет сравнить не одно, а сразу два значения с их предыдущими значениями.

На современных процессорах сейчас есть операция, которая позволяет делать сравнение сразу двух operandов. Мы либо для всей структуры данных, либо для каждого элемента в отдельности храним число операций push() и pop(). Когда мы делаем pop(), мы запоминаем текущее значение этого counter'a всех инструкций pop() и значение для push(). Если оно на момент CAS изменилось, то процесс завершается с ошибкой. Если нет, то продолжает выполняться.

Мы боремся с проблемой АВА тем, что добавляем вспомогательную переменную, которая подсчитывает количество операций.

Данная операция работает с 16 байтами: 8 байт для адреса и 8 байт для counter'a. На современных архитектурах хранить counter'ы в 32 битах нехорошо. На интенсивных структурах они очень быстро исчерпываются. На 64 битах они будут пожизненными.



Второй вариант очереди — это очередь, реализованная на массиве. Здесь также есть проблема АВА, но мы работаем уже не с указателями в памяти, а с указателями элемента (с индексами в массиве). Мы должны поддерживать два индекса — голова списка и его хвост.

Чем хорош массив? Он имеет ограниченный размер. Соответственно, мы контролируем размер очереди. Плохая ситуация возникает, когда у нас worker'ы не успевают, и producer'ы напихивают очередь до того состояния, когда весь demon уходит в swap. Нам и так было плохо, все становится еще хуже. Из этого следует, что очереди и все структуры данных всегда нужно лимитировать по размеру.

В очереди на массиве лимит заложен в самой архитектуре — мы выделяем массив ограниченного размера. Для списка нам нужно будет держать вспомогательные переменные, содержащие текущий размер списка.

Поскольку мы должны контролировать размер, то мы должны отслеживать все потоки, которые добавляют и извлекают из очереди. С операцией pop() все достаточно плохо. Ячейка на картинке соответствует восьми байтам, которые в реальности являются указателем. Мы в очередь просто напихиваем указатели.

Когда мы делаем pop() из очереди, он нам возвращает значение, которое извлек, то есть возвращает указатель. Мы должны атомарным образом контролировать, чтобы уже извлеченные из очереди указатели не перезаписывались.

На картинке показано, что массив находится в заполненном состоянии. У нас есть воображаемая вторая половина этого массива. Head начинает догонять tail. Tail прошел в середину массива, head начинает догонять сначала. Нам нужно контролировать последний индекс в очереди (tail1, красным на рисунке), который был извлечен. Пока поток, который считал значение из этой ячейки, не считает новое значение, мы не имеем право его трогать.

Кроме указателей head и tail нам нужно в очереди удерживать указатели для каждого producer'a и consumer'a, для каждого потока, который работает с этой очередью. В данной ситуации у нас два producer'a и два consumer'a. Мы держим два вспомогательных tail и два вспомогательных head. Это позволяет контролировать ситуацию, не допускать переполнения очереди и случаев, когда head догоняет tail.

Я уже говорил, что 64 бита — это очень много. Мы можем атомарно инкрементировать эти индексы (`_sync_fetch_and_add(&a, 1)`) и не беспокоиться о том, что нам нужно брать по остатку на деление. Когда мыдвигаемся по очереди, мы просто увеличиваем индекс и не беспокоимся о том, что он когда-нибудь переполнится. Это достаточно удобно.

hi⁺⁺ HighLoad⁺⁺

Lock-free очередь (список vs ring-buffer)

- Ring-buffer сложнее в реализации (требуется синхронизированное передвижение указателей на tail и head для каждого из потоков)
- Список должен быть интрузивным для избежания аллокации узлов на каждой вставке
- Для списка нужно отдельно реализовать контроль числа элементов
- Локализация и выравнивание памяти - ?

Сравним эти два подхода. Ring-buffer оказывается сложнее. Обычная очередь на списки опирается с двумя указателями на динамическую память. Она должна быть сложнее, так как указатели менее консистентны. Когда мы работаем с массивом, нам проще работать с обычными целыми числами, чем с указателями.

На практике из-за того, что требуется держать несколько индексов для `tail` и `head`, нам приходится сильно усложнить ring-buffer. Реализация такой очереди становится достаточно своеобразной.

Если мы используем очередь в виде списка, мы должны позаботиться о том, как мы будем добавлять новые элементы. Классический список — это С-шная структура, которая держит в себе указатели на `next` и элемент, который соответствует данной структуре.

Здесь стоит заметить, что с помощью lock-free мы можем организовать только односвязную очередь. Двухсвязная очередь никогда не будет lock-free.

Если каждому элементу списка нам нужна область памяти, то мы должны выделять ее. Часто приходится выделять ее динамически. Звать на каждой операции с очередью `malloc` и `free` — это очень дорого. Можно использовать кэш типа SLAB-аллокатора. Другой вариант — наша очередь должна быть интрузивной. Это значит, что указатели на `next` должны содержаться в самой структуре, которая хранится в очереди.

Про контроль количества операций я уже говорил. Еще один интересный вопрос — локализация памяти. Если у нас очередь большая, элементов много, то кэш процессора будет использоваться достаточно активно. Кэш L1 будет вылетать весь. Это значит, что в структуре данных следует как можно лучше локализовать память. Должно быть как можно меньше обращений к указателям, удаленным друг от друга.

Непонятно, в какой из этих структур данных будет лучшая локализация памяти. Что одна, что другая так или иначе взаимодействуют с указателями.



HighLoad++

Lock-free очередь (только)

- Работает только для очередей — сканировать такие структуры данных без блокировок нельзя
- Для очереди нужна реализация ожидания:
 - usleep(1000) — помещение потока в wait queue на примерно один такт системного таймера
 - sched_yield() - busy loop на перепланирование (100% CPU usage)

Еще раз акцентирую внимание. Lock-free — это всегда односвязная (LIFO или FIFO) структура данных, на которой определены только две операции — pop() и push(). Мы не можем сканировать эти структуры данных (не можем искать никакие элементы), не можем делать sort и прочие вещи. Если нужно сделать что-то сложнее push() и pop(), мы должны использовать блокировки.

Мы должны контролировать не только число элементов в списке, но и ожидание. Допустим, у нас кончилось место в очереди или элементов еще нет. Поток, который кладет новый или достает существующий элемент, должен перейти в состояние ожидания. Это состояние может вызываться достаточно часто.

У нас есть две альтернативы: либо поспать, либо сказать планировщику, что потоку процессор не нужен. В последнем случае он может перепланировать на процессор другой поток.

Первый вариант — это usleep. Usleep(1000) хорош тем, что это всего 1/1000 секунды. В современном Linux 1/1000 секунды — это один такт планировщика. Мы уходим в ядро, текущую задачу помещаем в очередь ожидания. На следующем такте задача пробуждается. Выполняется некоторая работа ядром системы, но при этом мы делаем то же самое, что ожидаем от shed_yield().

Shed_yield() — это busy loop. Если вы его будете использовать, вы увидите 100% использования процессора. Планировщик будет перепланировать эту задачу снова и снова.



HighLoad++

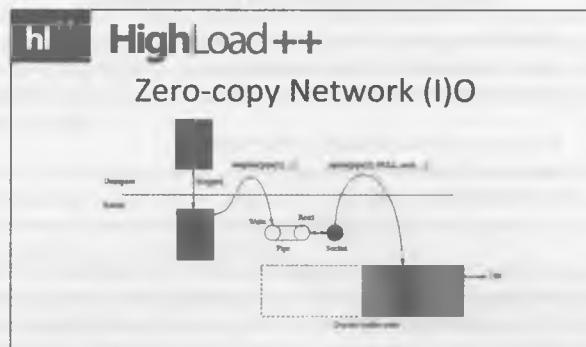
Кэш

- Hash table, RB-tree, T-tree, хэш деревьев и пр.
- Lock contention снижается путем введения отдельных блокировок для каждого bucket'а хэша или поддерева
- Часто нужно вытеснение элементов: Hash table как список или список + основная структура данных
- RW-блокировки, ленивые блокировки

Я уже говорил, что кэш так же критичен к Lock contention. Здесь мы не можем использовать атомарную структуру данных. Единственное, что мы можем делать, — гранулировать блокировки. Обычно это стандартная структура данных — дерево или хэш-таблица. Мы используем отдельные блокировки либо для каждого bucket'а в хэше, либо для поддерева в дереве. Это дает очень хороший результат, параллельность сервера сильно возрастает. Если мы, например, для сервера в 16 потоков используем 64 блокировки, то результат будет очень хороший.

Часто на хэшах нужна операция вытеснения какого-то старого элемента с целью его замены на новый элемент. Для этого нам нужна либо вторая структура данных, которая будет держать устаревшие элементы, либо можно использовать хэш-таблицу, которую можно сканировать как список. Для таких хэшей я рекомендую использовать хэш-таблицу. Она очень

удобна. Это двухуровневый список, который можно пробежать либо по всем bucket'ам, либо по каждому списку. Каждый bucket можно нагружить reference counting'ом, timestamp'ом (последним обращением) и так далее.



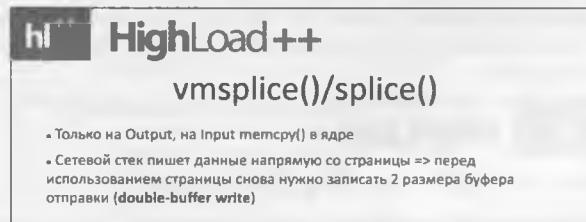
Несколько слов про Zero-copy. В Linux это достаточно новая технология. Во FreeBSD она появилась намного раньше.

В Linux она работает только на вывод. Вы алоцируете страницу памяти (показана коричневым в user space), она видна ядру, то есть она mmap()'ится в область пространства ядра. Вы должны завести Pipe. У него два конца: один на чтение, другой на запись. Плюс socket, в который хотите записать.

Vmsplice() map'ит страницу на Pipe, то есть кладет указатель на эту страницу в буфер Pipe. Splice() перемещает страницу из Pipe в буфер отправки протокола (TCP, UDP и любой другой прикладной протокол).

Когда мы выполнили два системных вызова, мы поместили данные в очередь отправки. Мы не знаем, когда в реальности данные будут отправлены. Мы завершили системные вызовы, но данные еще не отправлены. Мы ничего не можем писать в эту область памяти. Если мы туда запишем, то мы запишем напрямую в буфер TCP или другого протокола.

Нам нужно быть уверенными, что страница, которую мы хотим использовать снова, была действительно отправлена. Для этого используется Double-buffer writing. Мы записали страницу, потому выделяем область памяти, равную буферу отправки, и пишем его в сокет. Только теперь, когда у нас точно через сокет прошел объем данных, раный его буферу, мы можем использовать эту страницу снова.



Vmsplice() и splice() можно использовать для приемки. Они будут нормально работать, но у них не будет никакого преимущества перед обычным read(2) (скорее, даже наоборот из-за того, что теперь делается два системных вызова вместо одного). Они будут реализованы через memcopy() внутри ядра.

hl++

HighLoad++

Splice: производительность

```
# ./nettest/xmit -s65536 -p1000000 127.0.0.1 5500
xmit: msg=64kb, packets=1000000 vmsplice() -> splice()
usr=9259, sys=6864, real=27973

# ./nettest/xmit -s65536 -p1000000 -n 127.0.0.1 5500
xmit: msg=64kb, packets=1000000 send()
usr=8762, sys=25497, real=34261
```

На этом слайде небольшие измерения производительности для splice(). Нам нужно смотреть на время sys. Реальное время примерно одинаково, sys здорово отличается. Реальное время одинаково из-за того, что тесты проводились на localhost, мы упирались в буфера отправки socket'a. Если посмотреть на Send-Q в netstat, видно, что буфера заполнены, поэтому время ожидания большое.

Sys — это то место, где выполняется копирование данных из user space в kernel, поэтому для нас важно именно это время.

hl++

HighLoad++

Когда полезны свои аллокаторы

- Специальный аллокатор, позволяющий читать из сокета по несколько сообщений и освобождать весь кусок разом (страницы + referee counting)
- Page allocator для работы с vmsplice/splice
- SLAB-аллокатор для объектов одинакового размера
- Boost::pool (частный случай SLAB-аллокатора): пул объектов одинакового размера, освобождаемых одновременно

Как мы уже говорили, если мы используем Zero-copy, мы должны сделать какие-то подпорки на аллокации данных, на использовании страниц. Нам нужен page allocator, потому что splice() работает только с теми данными, которые измеряются страницами. Нужно контролировать, когда эта страница будет освобождена. Нужен специальный аллокатор памяти. Архитектура сервера получается завязанной в клубок: потоки, ввод-вывод и аллокатор памяти, которые неотделимы друг от друга.

Помимо Page allocator есть SLAB-аллокатор. Это аллокатор памяти, который оптимизирован под выделение объектов равного размера. Boost предоставляет специализацию SLAB-аллокатора, который позволяет не задумываться об освобождении объектов. Мы можем освобождать их одним куском.

hl++

HighLoad++

CPU binding (interrupts)

- APIC балансирует нагрузку между свободными ядрами
- Irqbalance умеет привязывать прерывания в зависимости от процессорной топологии и текущей нагрузки
=> не следует привязывать прерывания руками
- Прерывание обрабатывается локальным softirq, прикладной процесс мигрирует на этот же CPU

```
Cpu9 : 13.3%us, 62.1%sy, 0.0%ni, 1.0%id, 0.0%wa, 0.0%hi, 23.6%si, 0.0%st
Cpu10 : 0.0%us, 0.7%sy, 0.0%ni, 82.7%id, 0.0%wa, 0.0%hi, 16.6%si, 0.0%st
```

Коротко про CPU binding. У нас есть несколько binding'ов — binding прерываний, binding прикладных процессов. Прерывания сейчас хорошо планируются на современных архитектурах с помощью APIC. APIC перераспределяет прерывания на наиболее свободный процессор. Вообще говоря, как именно будут планироваться прерывания по процессорам, даже если все процессоры будут указаны как доступные в сри mask, зависит от конкретного железа. Часто, даже если APIC может отправлять прерывания на несколько процессоров, он отправляет их только на первый. Делается это из соображений лучшего использования процессорных кэшей обработчиками прерываний..

Снизу на слайде есть статистика очень нагруженного процессора, который отправляет много данных на другой процессор. На этом (первом) процессоре висит user space процесс, softirq и прерывания. Мы уже не справляемся на этом ядре и нам приходит на помощь другое ядро - softirq и прерывания частично уходят на него

В Linux есть демон irqbalance, который автоматически привязывает прерывания. Он делает это в зависимости от текущей нагрузки и типа прерываний.

- Не имеет смысла создавать больше тредов, чем физических ядер процессора для улучшения cache hit
- Часто кэши процессора разделяются ядрами (L2, L3)

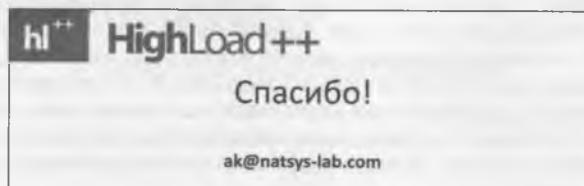
=> worker'ов (разделяющих кэш) лучше привязывать к ядрам одного процессора.

Шины данных между ядрами не одинаковы. Шина между ядрами одного процессора намного быстрее, чем шина данных между разными процессорами. Ядра одного процессора часто используют общий кэш L2 или L3. Если мы реализуем сервер с worker'ами, то их выгоднее привязывать на один процессорный пэкдж, чтобы увеличить скорость работы с разделяемыми структурами данных.

```
# dd if=/dev/zero count=2000000 bs=8192 | nc 10.10.10.10 7700
16384000000 bytes (16 GB) copied, 59.4648 seconds, 276 MB/s

# taskset 0x400 dd if=/dev/zero count=2000000 bs=8192 \
| taskset 0x200 nc 10.10.10.10 7700
16384000000 bytes (16 GB) copied, 39.8281 seconds, 411 MB/s
```

Пример производительности. Первый случай — без привязки к процессорам. Второй — привязка к ядрам одного процессора. Производительность при этом намного выше.



Спасибо.

Pconnect: граната в руках обезьяны

Сергей Аверин



HighLoad++

Pconnect: граната в руках обезьяны

Сергей Аверин

Ведущий: Следующий доклад от Сергея Аверина из компании «Badoo». Устраивайтесь поудобнее. Говорят, будет очень грандиозное шоу. После доклада Сергея Аверина будет доклад Андрея Аксенова. Обещает быть не менее интересным. У нас сегодня отличная подборка докладчиков в нашем зале.

Сергей Аверин: Здравствуйте.



HighLoad++

badoo — это:

Социальная сеть для знакомств с новыми людьми
В Top-200 Alexa с 2007 года
127 миллионов зарегистрированных пользователей
10 миллионов пользователей в день
2 миллиона фотографий загружаются ежедневно

Сегодня я буду пугать вас persistent connect'ами, и только настоящий самурай сможет освоить persistent connect'ы и победить их. У всех остальных я просто сожру мозг.

Ну ладно, ближе к делу.

Я работаю в компании «Badoo». Это смесь социального сайта, сети знакомств, «фотошарингового» сервиса. Мы достаточно крупный сайт, находимся с 2007-го года в Top-200 Alexa. У нас порядка 127-ми миллионов пользователей. Пользователи сайта в день загружают порядка, по-моему, двух миллионов фотографий. DAU у нас в районе десяти миллионов пользователей в день.



HighLoad++

badoo — это:

30 тыс. запросов/сек к PHP backends
MySQL, PHP, C/C++, Linux, nginx, PHP-FPM, memcached
Много своего

С точки зрения программиста это 30 тысяч запросов в секунду к backend'ам. Ясное дело, что на 1 request может выполняться еще несколько подзапросов. Мы используем много open-source софта — MySQL, PHP, PHP-FPM. Также много C/C++ демонов.

hl++ HighLoad++

Pconnect — что это?

Persistent connection/keep-alive/connection reuse — по сути одно и то же
Pconnect в мире FastCGI-демонов

Плюсы:

- Экономия ресурсов сервера
- Меньшая нагрузка на сеть
- Экономия времени и ускорение отклика

Про persistent connect'ы. Что это такое. Еще иногда называют keep-alive или connection reuse. По сути, это одно и то же, хотя есть небольшая разница. Это не технология — это методика использования соединения для посылки нескольких команд и получения нескольких ответов в одном соединении. На самом деле, изначально все так и работали. Просто в мире веба появилось это «кликовое сознание», появилась идея, что один request — один connect, потом закрыли — открыли новый.

В мире FastCGI-демонов это значит, что... Допустим, на примере PHP-FPM. У вас есть несколько процессов-обработчиков. В каждом процессе — один тред. Этот тред будет хозяином, держателем ваших persistent-соединений и будет передавать их между запросами.

В один конкретный момент времени в одном обработчике будет обрабатываться только один веб-request. Ваше соединение, которое вы открыли, передастся следующему запросу только после того, как текущий полностью закончится (это очень важно).

Какие плюсы нам дает эта методика:

- Экономим ресурсы сервера, потому что connect — это дорогая операция, нужно переслать 3 пакета.
- Экономим время, потому что нам нужно переслать 3 пакета, дождаться подтверждения (сеть может быть медленной), а так мы считаем, что соединение у нас всегда открыто.
- Конечно, меньше нагрузка на сеть, на ядро, на память и на сетевую подсистему.

hl++ HighLoad++

Действительно Plug'n'Play?

<http://www.php.net/manual/en/features.persistent-connections.php>

"Persistent connections were designed to have one-to-one mapping to regular connections. That means that you should always be able to replace persistent connections with non-persistent connections, and it won't change the way your script behaves."

Это маркетинг. И не более того.

Во всех мануалах, в которые я залазил и читал, когда стал близко с этим работать, была такая идея (особенно в PHP хорошо написано). Persistent-соединения были придуманы как plug-in замена. Просто меняйте функции mysql_connect, например, на mysql_pconnect — и все начинает летать-работать, появляются волшебные феи, и ваше приложение ускоряется в 3 раза. На самом деле, все ни фига не так. Я утверждаю, что это маркетинг и не более того.

hi⁺⁺ HighLoad++

В реальности

Чаще всего не Plug'n'Play

Должны быть рассчитаны на reconnect:

- протокол
- клиентская библиотека
- серверное ПО

Не «серебряная пуля»

В больших/highload системах более ощутимо

Во-первых, на опыте моего проекта выяснилось, что чаще всего это не Plug'n'Play. Очень важно, что на persistent-соединение должен быть рассчитан протокол, по которому вы общаетесь, ваша клиентская библиотека, клиентский код и серверное ПО зачастую тоже.

Во-вторых, это ни фига не «серебряная пуля». Я даже скажу, что увеличивается не количество ошибок, а увеличивается гораздо сильнее вероятность ошибок и падений.

В highload-системах при больших нагрузках и большом количестве запросов просто по закону больших чисел вы столкнетесь с проблемами гораздо раньше, и последствия от них будут гораздо более печальными.

hi⁺⁺ HighLoad++

Проблемы

1

Софт не рассчитан на большое кол-во открытых соединений

Сколько памяти съедает инициализированный коннект?

Выделяется ли в софте по треду/процессу на коннект?

Есть ли ограничение на кол-во коннектов с одного адреса?

Ограничения сетевой подсистемы Linux

Теперь немного теории. Какие проблемы мы классифицировали по моему опыту.

Во-первых. Банальная проблема. Ваш софт, с которым вы общаетесь, может быть просто не рассчитан на большое количество соединений. Например, один инициализированный коннект съедает кучу памяти. Или просто открытие коннекта приводит к большой нагрузке на сервер (допустим, держание коннекта).

Дальше. Есть некоторые демоны, которые выделяют по процессу на коннект. Это значит, что Linux'овое ядро рано или поздно умрет, если вы попытаетесь открыть 10 тысяч постоянно висящих соединений, потому что у вас будет 10 тысяч процессов.

Дальше. Существуют вполне конкретные ограничения ядра Linux на количество открытых исходящих сокетов с одного IP-адреса. Существуют вполне конкретные буферы, которые не бесконечны. Есть ограничения на количество одновременно открытых файлов в процессе (сокет там тоже является файлом, насколько я понимаю).

Дальше. В некотором софте вводится принудительное ограничение на количество коннектов с одного адреса. Это сделано как раз для того, чтобы экономить треды/процессы и чтобы вы вот так не игрались. Поэтому в первую очередь посмотрите, что за софт вы используете, выдержит ли он большое количество одновременно открытых соединений.

hl++ HighLoad++

Проблемы

2

Разберитесь в коде

Не закрывайте хорошие соединения функцией `fclose`
Проверьте, как работает ваш extension, особенно все его warning'и и error'ы

- <http://www.php.net/manual/en/memcached.addservers.php>

В php вам поможет `fseek($socket)`

Во-вторых. Вообще удивительная ситуация, которая меня радует. Функция `fclose` закрывает persistent-соединение, а функция `mysql_close` не закрывает persistent-соединение. Мало того, во многих PHP extension'ах вообще нельзя закрыть соединение, если оно persistent. Оно просто делает вид, что оно закрылось. На самом деле, нет.

Я, во-первых, предлагаю вам проверить, как работает ваш extension. Особенно все его warning'и и error'ы. Научитесь детектировать, открылось ли соединение новое или было persistent-соединение. Разберитесь с тем, какие валятся ошибки. Вполне вероятно, что при каком-то типе ошибок соединение нужно принудительно закрывать (и это так).

Самая глупая вообще ошибка, на которую я натыкался. Во встроенному в PHP memcached и клиентской библиотеке есть функция `add_servers`, которая добавляет в некий пул сервера. Если у вас свежеоткрытое соединение, то вы добавите сервера в пустой список. Допустим, добавили 3. В следующий раз это же соединение, когда вы получите и опять попытаетесь 3 добавить, они опять добавятся. У вас пул был 3, потом пул стал 6, потом будет 9. Он так будет бесконечно расти в памяти.

Там нужно писать идеальный код, который будет проверять, открыто ли это persistent-соединение или нет и добавлять или не добавлять (в зависимости от того, что добавлено в этот пул) сервера заново. Аккуратно относитесь к коду.

hl++ HighLoad++

Проблемы

3

Когда нужно закрывать соединения

- При синтаксических ошибках
- При тайм-аутах
- Если вы не дочитали ответ до конца

В-третьих. Самая, на мой взгляд, важная проблема. Дело в том, что когда вы посыпаете запрос на чтение через какой-либо сокет, и в ответ вам возвращается ответ, ваш софт должен уметь нарезать эти ответы как-то на кадры. Понимать, что пришла одна команда: вот ее начало, вот ее конец.

Зачастую (а я с этим сталкивался) в некоторых демонах бывают проблемы. Например, лишние байты появляются. Например, байты бьются. У меня есть демон, который периодически бьет байты. Проблема в том, что если у вас обычное соединение, вы получили ошибку, то вы сейчас закроете коннект, откроете его заново, повторите запрос на чтение — отлично, у вас все придет.

В persistent-соединениях совершенно другая ситуация. Вы послали запрос на чтение, вам в ответ пришел битый кадр. Вы прочитали его половину, считая, что байт побился или приш-

ли нули. Вы прочитали 00, потом начался какой-то текст. От этого текста взяли первую букву – будите считать, что это длина команды. Отрезали кусок команды, а в сокете еще лежит недочитанный кусок. Вы получите сейчас ошибку, вы не сможете распарсить синтаксис этого ответа.

Вы пошлете следующий запрос на чтение в этом же соединении (возможно, даже в этом же web-request'e, возможно, в следующем). Вы получите второй недочитанный кусок текста – опять будет синтаксис error, и так далее.

Получается так, что нарезка на эти кадры ответов у вас бьется. Если у вас persistent-соединение, то вы получаете полосы – так называемый «эффект трека», когда в видео-сигнале получается шум на несколько кадров вперед, потом опять появляется синхрокадр и дальше идет нормальная телекартинка.

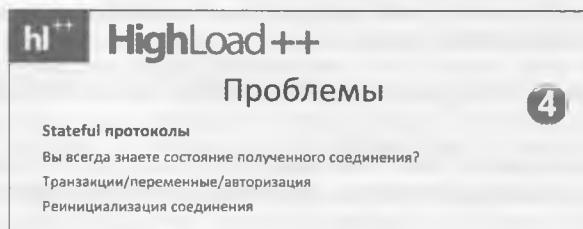
Когда у вас происходит синтаксическая ошибка, вы всегда должны написать софт, который будет ее детектировать и правильно восстанавливать структуру ваших ответов, команд (то есть вырезать команды из этого бинарного потока).

Либо более дешевый вариант – закрыть соединение и открыть новое. Иначе вы получите полосы. Ваш сокет будет передан следующему обработчику, он опять пошлет команду на чтение или на запись, и в ответ опять получит битый ответ. У вас будет такая бесконечная полоса до тех пор, пока вы не исчерпаете количество запросов (в PHP-FPM в районе 1 тысячи, по-моему, стоит обычно), после которых полностью гасится этот поток и форкается новый.

То же самое касается тайм-аутов. Если вы послали запрос на чтение, и у вас сработал тайм-аут, это значит, что ответ придет вашему потомку (тому, кто будет обрабатывать соединение после вас). У меня это привело к тому, что мы один раз залогинили 10 тысяч пользователей не в свои аккаунты.

Схема была очень простая. В этом коннекте были только команды на чтение. Но ответы всегда были одинаковые. Грубо говоря, select * where id= и дальше шло число. В ответ мы иногда при тайм-ауте получали два ответа – от предыдущего запроса и от текущего. Поскольку мы не проверяли вообще ничего про это дело, то получилось так, что мы считали предыдущий ответ валидным нам, и он был абсолютно синтаксически валиден. Поэтому пользователь залогинился в аккаунт предыдущего – того, кто авторизовался раньше.

То же самое, если вы недочитали ответ до конца. Хотя существуют протоколы, где можно бросить, и там будут недочитанные данные в сокете. Вы опять сами себе портите жизнь на будущее, потому что вы создаете трек – эффект сбитой синхронизации.



Дальше. Stateful протоколы. Что это такое. Протокол, у которого есть состояние, переменные, и прочее. Проблема тут простая. Когда вы получаете коннект от MySQL (persistent, которым кто-то раньше пользовался), вы всегда можете с гарантией сказать, открыта ли у вас транзакция, какие у вас переменные, какой текущий charset, какая текущая таймзона выставлена у этого коннекта? Зачастую нет.

Это значит, что вам нужно либо всегда работать с одной средой (постараться сделать так, чтобы state у этих коннектов всегда был идентичный и работать с тем предположением, что всегда все хорошо и всегда это состояние такое). Но, допустим, в транзакционных базах данных это вообще не работает. Вы можете открыть транзакцию, у вас будет тайм-аут, вы отдачите этот коннект с открытой транзакцией следующему обработчику, и он будет работать в режиме открытой транзакции, сам на это не рассчитывая.

Именно поэтому в некоторых хороших базах данных есть команда на реинициализацию соединений, то есть делание вида, что коннект открылся заново. В MySQL это делается с помощью авторизации юзера.

Посмотрите обязательно, пожалуйста, в PHP есть два extension'a: mysql и mysqli. MySQLi имеет режим, когда persistent-соединение реинициализируется. Оно вызывает эту функцию. Если у вас другой язык программирования, у вас своя клиентская библиотека, обязательно разберитесь со state'ом. Сделайте так, чтобы вы всегда гарантированно точно знали или сами вручную приводили коннект в то состояние, которое вам нужно. Иногда в MySQL (но это не очень правильно, я сразу скажу) делают rollback в начале получения каждого соединения.

The screenshot shows a section from a HighLoad++ article. At the top, there's a logo 'hl ++' followed by 'HighLoad++'. Below it is the title 'Проблемы' (Problems) with a large number '5' in a circle to its right. Under the title, there's a sub-section header 'Асинхронная природа сокетов' (Asynchronous nature of sockets). A note below it says 'Должна быть возможность сопоставления запросов и ответов' (There must be a way to correlate requests and responses). A bulleted list follows:

- Единственное, где я это нашел, это в MongoDB и MessagePack-RPC
 - <http://www.mongodb.org/display/DOCS/Mongo+Wire+Protocol>
 - <http://wiki.msgpack.org/display/MSGPACK/Design+of+RPC>

At the bottom of this section, it says 'Способ, которым мы пользуемся (не дает гарантии)' (The way we use it (does not provide guarantees)).

Дальше. Самое зло сокетов. Дело в том, что сокеты (как и все в природе) асинхронны. Никакой гарантии, что вы получите ответ в ближайшую 1 секунду, 2, 5, 10, нет. Мало того, нет никакой гарантии, что вы сейчас спросили демон — и вам в ответ придет ответ на ваш запрос. Никакой гарантии. Может быть, это ответ на запрос, который он обрабатывал 10 секунд, и он долго-долго его обрабатывал и теперь выплюнул вам ответ. Это не вы спрашивали. Протокол должен быть рассчитан на rconnect'ы, иначе у вас будут большие проблемы.

В протоколе должна быть возможность сопоставления запроса и ответа. Это значит, что вы послали команду, в ней какой-то рандомный id'шник вписан, и ответ вам возвращается с этим рандомным id'шником. Реализовать это можно по-разному. Но единственный протокол, где это вписано прямо исконно, это протокол MongoDB (за что я ребят уважаю). Может быть, база спорная, но MongoDB отлично работает с rconnect'ами. Мало того, в PHP extension'e все коннекты persistent, и отлично работают.

Второе — это мало кому известная альтернатива Google Protobuf. Называется MessagePack-RPC.

Способ, которым мы пользуемся. Поскольку нам нужно было все это внедрять, делать было нечего, были вполне очевидные проблемы, что «вот, залогинили каких-то идиотов не туда, они жалуются».

Мы делали очень хитро. Мы делали запросы на чтение так, чтобы нам вернулись и ключ, и данные. У нас была Key-value база данных. Мы делали select key, data, where key=1 и проверяли, что в ответе key соответствует тому, который мы просили. В случае если не соответствует, мы закрывали коннект, открывали заново и повторяли запрос. Это не дает стопроцентной гарантии, но это «солома», которая позволит вам ночью спокойно спать.

hl⁺⁺ HighLoad++

Проблемы

6

`set_time_limit`

Есть, было и будет в FastCGI-демонах

Терминация запроса пользователем

Shutdown обработчик

Дальше. Вполне очевидная PHP-шная проблема, про которую все забывают. Во-первых, существует мало кому известная терминация запроса пользователем в PHP. Вы можете в браузере, пока он грузится, нажать кнопку «стоп». Этот сигнал придет в ваш PHP FastCGI-демон и будет им обработан. Причем вы с этой обработкой можете работать по-разному. По дефолту она вызывает все-таки Shutdown обработчик, так что тут PHP-шники сами себе солому подстелили. Все будет хорошо.

Но есть таймер. Таймер на 30 секунд по дефолту, который как раз позволяет вам делать так, чтобы у вас не было бесконечно исполняющихся FastCGI request'ов. Для этого в PHP встроен какой-то таймер. Для этого там есть возможность назначить callback на этот таймер. Он обычно вызывается, все хорошо работает.

Так вот. Вы должны отслеживать все свои persistent-соединения в каждом процессе, в некоем пуле, с которыми вы работали. Вы должны эти соединения делить на две группы. Те соединения, с которыми сейчас непонятно состояние и они условно плохие. Там послан запрос, ответ еще, допустим, не был получен. Также те соединения, которые хорошие, с которыми проблем не будет. Плохие соединения в функции-обработчике закрывать, иначе у вас произошел таймаут, вы передадите эти плохие соединения дальше и опять вызовете себе полосы сбивки или чтение не своих ответов. Про это все забывают.

hl⁺⁺ HighLoad++

Проблемы

7

Ограниченный функционал FastCGI-демонов

Клиентские библиотеки зачастую бездумно написаны

Не всегда можно понять новое ли это соединение

Нет полноценных connection pool'ов

Дальше. Достаточно «троллевская» тема. Я сразу скажу, что тут, наверное, спорно, но на мой взгляд... Я внедрял `rconnect`'ы в 3-4-х совершенно разных протоколах. Клиентские библиотеки очень бездумно написаны. Библиотеки и протоколы, рассчитанные на persistent-соединение прямо из коробки, хорошо, качественно и, главное, с гарантией (с гарантией, приближающейся к гарантии совпадения двух чисел CRC32), реализованы только в MongoDB и MessagePack-RPC. Во всех остальных солома либо вообще не подстелена...

Например, стандартный старый `memcached`'а, с которым я работал. Даже не различал ошибки синтаксиса от ошибок «ключ не найден». Он просто говорил «`false`» – и все. Это могло означать что угодно. Это могло означать, что соединение порвалось. Это могло означать, что пришел ответ, но синтаксис в нем битый, мы не смогли его прочитать. Это могло означать, что ответ пришел правильный, а ключа в `memcached` нет.

Нужно эти ошибки всегда классифицировать. На те ошибки, которые касаются синтаксиса и синхронизации, обязательно закрывать соединения или предпринимать меры самому. Но предпринимать меры самому можно, в основном, только в самописных библиотеках. Про это позже.

Во-вторых, не всегда можно понять, новое ли это соединение (это нужно, я потом буду рассказывать почему). Это полезная, важная фича. Но в большинстве extension'ов вообще нельзя это сделать.

В-третьих, нет полноценных connection pool'ов, то есть нельзя аккуратно регулировать ресурсы. Про это я тоже буду рассказывать. Нельзя ограничить количество этих persistent-соединений. Нельзя подоставать, позакрывать. Нельзя посмотреть время их жизни, когда они были открыты, что да как.

hl ++ HighLoad++

От теории к практике

От теории к практике.

hl ++ HighLoad++

Badoo Desktop

- Бесплатная Win/Mac программа
- Поддерживает ваш онлайн-статус на сайте
- Уведомления о новых событиях
- Дает нам местоположение пользователя
- Удобный доступ к разделам badoo.com

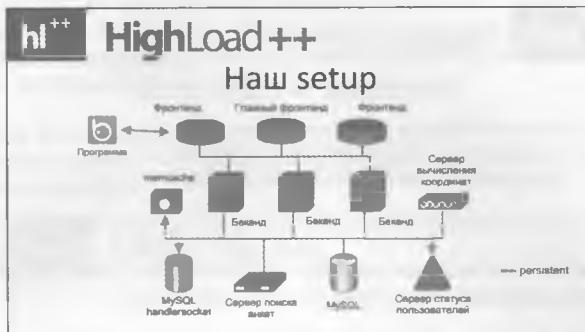
Я являюсь руководителем разработки приложения, которое называется Badoo Desktop. Оно есть под Windows, есть под Mac, сидит в Tray'e и сильно похоже на нотификатор, которым, по сути, является. Помогает вам удобно попасть на разделы сайта, уведомляет вас о новых событиях, дает нам информацию о вашем местоположении.

hl ++ HighLoad++

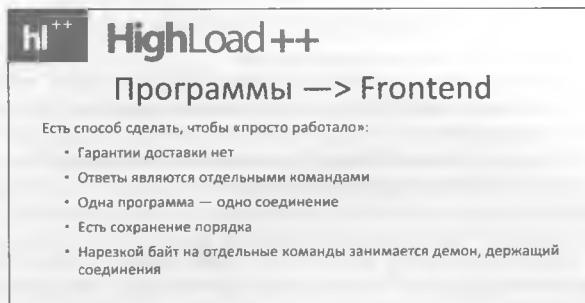
Badoo Desktop

- 3 млн. пользователей в месяц
- 1 млн. подключенных программ в пике
- 20 тыс. запросов/сек к PHP backends

С точки зрения программиста: 3 миллиона пользователей в месяц, 1 миллион постоянно открытых подключенных pconnect'ов, 20 тысяч запросов к backend'ам.



Такой у нас setup на площадке. Программа коннектится к фронтенду. Фронтенд рандомно раскидывает на один из PHP-шных бекендов, который уже в свою очередь ходит в memcache, в базы, в поиск анкет. Где зеленая стрелочка на слайде, мы внедрили persistent-соединения. Почему — давайте рассмотрим подробно.



Во-первых, из программы до frontend'a. Здесь задача была очень простая. Она звучала как «нужно ускорить время доставки нотификации», чтобы пользователь 15 минут не ждал, что ему пришло новое сообщение, получал ответ сразу. Мы подошли совершенно другим путем. Не стали делать гарантию, не стали делать сложных синхронизаций. Мы сами себе сказали, что гарантии доставки сообщений нет. Каждый ответ является, по сути, отдельной командой, поэтому сопоставления запросов с ответами нам не нужно.

У нас, по сути, получается архитектура а-ля сервер-сервер. Программа на сервер гонит команду на выполнение каких-то действий. Сервер в программу гонит команду на выполнение каких-то действий. Ответов, по сути, нет вообще. Это редкий случай, но... Так что вы можете совершенно по-другому извлечь эту проблему вот так оригинально.

Проблемы синхронности, параллельности и прочие решаются тем, что одна программа — одно соединение. Программа сама всегда отслеживает, чтобы коннект был ровно один. Есть сохранение порядка команд. В каком порядке, в котором послали, в таком порядке и получили.

Дальше с серверной стороны (чтобы не было проблем с сокетами, с вычитыванием и прочим, с закрыванием коннекта) нарезкой байт на отдельные кадры команд занимается С-шный демон, который держит постоянно этот rconnect. Он почти никогда не падает, он работает очень хорошо. Поэтому с точки зрения PHP никакой проблемы нет. Ему всегда приходит четкая команда, четкий один кадр.

hl ++ HighLoad++

Frontend —> Backend

Отдельный http-запрос на каждую входящую команду

Pconnect пока нет, но планируем через HTTP Keep-Alive

http://en.wikipedia.org/wiki/HTTP_persistent_connection

Соответственно, этим же демоном мы решили проблему того, как на трех серверах держать миллион с лишним одновременно открытых соединений.

С frontend'a к backend'у. То есть от С'шного демона до PHP-FPM. Отдельный http-запрос, в котором тело запроса бинарное, является командой, которая пришла из программы. Persistent-соединений пока нет, но планируем внедрить через HTTP Keep-Alive.

hl ++ HighLoad++

Backend —> MySQL

Протокол на pconnect полностью не рассчитан

Есть packet number, но оно обнуляется с каждым новым запросом

Есть реинициализация соединения, используйте mysqli_connect

Тред на коннект

Для нас малоинтересно, баз много, php-серверов много, запросы редки

http://forge.mysql.com/wiki/MySQLInternals_ClientServer_Protocol

Backend к MySQL. У нас очень много баз. Их — я не знаю — за 200 штук. Соединения с ними редки, запросы в них редки. Поэтому мы посчитали, что никакого смысла создавать себе геморрой с persistent-соединениями нет. Это накладно. Будет много постоянно открытых соединений до баз, которые и так нагружены. До них и так много соединений. Поэтому здесь мы решили не экономить «на спичках» и с persistent-соединениями морочиться не стали.

Протокол MySQL я изучал. На persistent-соединения он не рассчитан почти никак. Но в последних версиях (по-моему, в 5.1) появилась, наконец, нормально работающая реинициализация соединений, которая разорвет вам коннект и делает синхронизацию того, что вот, сейчас все, вы авторизовались, значит, у вас дальше соединение новое, свежее. Никакого непредсказуемого состояния в нем нет.

В MySQL один тред на коннект, поэтому количество коннектов в MySQL всегда ограничено. Вы это знаете прекрасно. Поэтому здесь я бы вам экспериментировать не советовал, если у вас много баз и много backend'ов. Используйте mysqli_connect с "r:", в котором как раз есть режим реинициализации, уже прямо защищенный в софте.

hl ++ HighLoad++

Backend —> HandlerSocket

Протокол на pconnect не рассчитан, зато коннект есть мало ресурсов

Используем pconnect

Написали свою библиотеку на php

Проверяя, что получили ключ, который просили

Получили прирост в 3—5 раз

<http://tinyurl.com/64vwn52>

HandlerSocket. Наверняка, кто-то знает. В прошлом докладе я рассказывал. Это такая попытка от SQL избавиться в MySQL. Протокол на rconnect совсем не рассчитан, но коннект ест мало ресурсов. Мы используем rconnect, применяя технику запроса и ключа, и данных и проверки, что ключ вернулся тот, который просили.

Библиотек было очень мало, одна падала в «корку», другая нам не понравилась. Написали свою. Дальше я буду рассказывать, какие даже бенефиты получили. Получили прирост в 3-5 раз на чтение. Отличные цифры.

hl⁺⁺ **HighLoad++**

Backend → Memcached

Протокол на rconnect не рассчитан, зато коннект ест мало ресурсов
Используем rconnect
Extension нормально работать не смог, написали свою библиотеку на php
Тщательно ловим все ошибки
Проверяем, что получили ключ, который просили
Получили прирост в 2 раза
<https://github.com/memcached/memcached/blob/master/doc/protocol.txt>
<http://code.google.com/p/memcached/wiki/MemcacheBinaryProtocol>

Memcached. Я изучал и текстовый протокол, и бинарную его версию. Они обе не рассчитаны на persistent-соединения. Однако мы используем rconnect, подстелили себе соломы. Extension нормально не работал, у него демон периодически плевался нулевыми байтами в ответе. В итоге у нас были постоянные 15-минутные пролежки из-за того, что memcached вообще не мог ничего прочитать, потому что каждая команда в ответе была сбитая, и мы ничего разобрать не могли. Написали свой extension, который умел вылавливать нулевые байты, обрезать этот кадр и работать дальше. Все стало прекрасно. Тщательно ловим все ошибки — вот в чем суть.

Также подстелили соломы — проверяем ключ. Ура! Это единственный протокол, в котором в ответе возвращается ключ, который просили. Получили прирост в 2 раза.

hl⁺⁺ **HighLoad++**

Backend → своя in-memory DB

Протокол на rconnect не рассчитан
Используем rconnect
Только команды на запись, ответ только либо OK либо ERROR
Гарантии записи нет и мы к этому готовы
Каждая отдельная ошибка нам не важна, они всегда идут только пачками

Backend — своя in-memory DB. У нас есть своя база данных, которая хранит статусы пользователя. Мы ее раз в минуту дергаем, говорим, что этот пользователь жив — и он на сайте становится online. Протокол на rconnect совершенно не рассчитан, он текстовый, очень простой и построен на базе протокола memcache. Однако мы используем persistent-соединения, не имея вообще никакой гарантии на то, что это будет хорошо работать, потому что у нас есть только запись, ответ бывает только либо OK, либо ERROR. Все команды однотипные. У них отличаются только значения user_id.

Гарантии записи нет, и мы на это рассчитаны. В случае больших проблем юзер не будет online 5 минут. Нам даже не нужна каждая отдельная ошибка, потому мы используем persistent-соединения. Нам важна полоса. Мы всех юзеров про отметили, потом опять про отметили. Если начинаются проблемы с демоном, то ошибка будет сразу в 15 тысяч в секунду. Я их увижу, ко мне придет мониторинг. Этот демон по одной ошибке никогда не плюет. Он либо работает, либо нет.

hl++ HighLoad++

Продвинутые подходы

Какой мы научились профит из этого извлекать, кроме тех профитов, про которые я рассказывал?

hl++ HighLoad++

Ленивая инициализация

В handlersocket-протоколе есть открытие индекса таблицы

Без rconnect открывали каждый раз после коннекта

С rconnect ловим ошибку про неизвестный индекс

Теперь мы можем открывать его только если это нужно

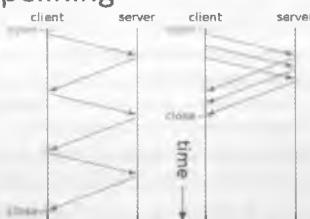
Во-первых, ленивая инициализация. В handlersocket, чтобы работать с таблицей, нужно ее открыть специальной командой `open_index`. Открываете `index`, с ним работаете. Без `rconnect` у нас, по сути, было 2 команды: сначала открывался `index`, потом шло чтение.

Сейчас мы используем более хитрый подход. Мы проверяем. Пытаемся читать. Если нам возвращается ошибка, что этот `index` закрыт, мы его открываем. Получается, что если раньше было 2 команды на запрос, то сейчас 1,0009, потому что эта команда выполняется на свеже-открытое соединение 1 раз — в тот момент, когда мы лезем в этот индекс. Даже если в этом соединении 10 тысяч индексов, мы их откроем только тогда, когда к ним будет реальная попытка доступа.

hl++ HighLoad++

Pipelining

Если в протоколе есть уникальный request-response id на каждый запрос, то будет надежно работать



Pipelining. Сложная штука. Бывает еще Parallel pipelining. Не буду про него рассказывать.

Суть очень простая. Если у вас есть уникальный `request-response` id, и ваш серверный демон может работать как-то в несколько процессов (то есть вы можете поставить 3 команды — он вам может вернуть 3 ответа), то используя эти `request-response`'ы, вы можете этот демон использовать. Вот так посыпать 3 запроса одновременно («Facebook», например, так делает — они в своих презентациях рассказывают). Они делают даже более интересно — они сначала пытаются высчитать все данные по пользователю, всех его друзей, все его фотки и прочее, а потом начинают проверять пермишны — мог ли он запрашивать эти фотки, нет.

Но поскольку 99% страниц, которые пользователи «Facebook» запрашивают, все-таки рендерятся (а 1% — это 503-я, 404-я ошибки), то у них получается так, что ответ приходит от разных-разных-разных демонов одновременно. Они вначале запроса послали 10-20 ответов, через секунду проверили — ok (через 10 миллисекунд). Все вернулось, отлично, сразу все данные обрабатывают, а не по очереди. У них код в этом плане несинхронный.

Используя request-response (сопоставление запрос — ответ) вы можете делать так, что у вас даже ответы будут возвращаться в разном порядке, и вы будете с ними читать, работать. Это, например, есть в handler-socket'е. Очень крутая штука.

The screenshot shows a section from the HighLoad++ documentation. At the top, there's a logo with 'hl' and 'HighLoad++'. Below it, the title 'Connection pooling' is centered. Under the title, there's a heading 'Плюсы:' (Pros) followed by a bulleted list:

- Уменьшение времени получения коннекта
- Упрощает код
- Дает контроль над использованием ресурсов

Below this, there's another heading 'Бывает внутри процесса и в виде отдельного демона' (Occurs inside the process and as a separate daemon). Another bulleted list follows:

- Внутри FastCGI — пишите сами, плюс у некоторых extension'ов есть свои пуллы. В мире Python — SQLAlchemy
- Вне процесса сложнее и функциональнее, обычно есть доп. логика

Connection pooling. Это создание некоего массива, который хранит все ваши коннекты. Бывает разный — двух видов. Внутри процесса или, допустим, внутри вашего FastCGI-демона, хотя таких я не видел никогда — всегда были внутри процесса. Либо в виде отдельной софтины, которая ставится на сервер и решает только эту задачу.

Плюсы.

- Уменьшаем время получения коннекта, потому что за нас это делает отдельная библиотека. У нас есть этот pool, он всегда доступен. Даже многие преинициализируют эти соединения (как только код или FastCGI-демон стартанул — сразу все коннекты открылись).
- Упрощает код очень сильно. Вы делаете вид, что «дай мне соединение, дай соединение, дай соединение», и все. Библиотека на себя берет много разного гемороя.
- Дает контроль над использованием ресурса, что на мой взгляд, очень интересно. Если у вас есть MySQL-база, вы к ней можете только к 5 коннектов входить, потому что у вас большое ограничение, то вы можете поставить себе софтину, которая будет держать только 5 коннектов, а их по очереди отдавать тем, кто просит. Главное — чтобы они отпускали этот коннект.

Внутри FastCGI ничего хорошего я не видел. В Java есть хорошее. Что касается PHP — пишите сами, у вас будет хороший навык. У некоторых extension'ов есть свои пуллы. Типа как в memcache, но там пул немножко другого плана. В мире Python'a — SQLAlchemy. Там это решено очень хорошо. Рекомендую.

Вне процесса обычно гораздо более функциональны софтины. Они решают еще кучу параллельных задач. У них есть дополнительная логика, и они вообще очень крутые. Редко кто, к сожалению, пользуется.

hl ++ HighLoad++

Connection pooling

pgpool — PostgreSQL connection pool daemon

- <http://pgpool.projects.postgresql.org/>
- В Postgres процесс на коннект

MySQL Proxy — швейцарский нож

- http://forge.mysql.com/wiki/MySQL_Proxy_FAQ
- load balancing, failover, query analysis/modification, R/W splitting

SQL Relay — ODBC, Oracle, MySQL, PostgreSQL, Sybase, MS SQL Server, IBM DB2, Firebird, SQLite and MS Access

- <http://sqlrelay.sourceforge.net/sqlrelay/faq.html>

Для Postgre'са. У него большая проблема с тем, что один процесс на коннект. Понимаете сами, в чем проблема. Используется pgpool.

Для MySQL используется обычно MySQL Proxy. Это вообще «швейцарский нож». Там, например, есть разделение чтения и записи по разным серверам, failover. Там есть логирование медленных запросов, query analysis. Всячески рекомендую.

Есть такая софтина, которую я никогда не видел, но народ хвалит. SQL Relay. Поддерживает вот такое большое количество баз данных. Пробуйте. Или напишите свой.

hl ++ HighLoad++

Чеклист

- 1) Что конкретно вы экономите и с какими последствиями?
- 2) Потянет ли это все ваша система?
- 3) Вы разобрались с кодом, ошибками, закрываете коннект когда нужно?
- 4) Рассчитан ли протокол на rconnect?
- 5) Если нет, вы подстелили себе соломы?
- 6) Вы правильно работаете со stateful протоколом?
- 7) Вы используете connection pool для экономии ресурсов?
- 8) Вы готовы писать свои велосипеды?

Чеклист. Мы хотим внедрить rconnect в highload-проекте. Ответьте себе на эти вопросы.

- 1) Что конкретно вы экономите? С какими последствиями примерно вы это экономите? Нафиг вам это вообще надо? Может, вы ничего не сэкономите. На что вы рассчитываете? Какой из трех этих гейнов, бенефитов вы хотите получить?
- 2) Потянет ли все это ваша система? Рассчитан ли ваш серверный софт, сеть, таблицы маршрутизации, роутер и вся-вся-вся эта фигня?
- 3) Вы разобрались, как работает ваш extension, проверили, что он с протоколом работает правильно, ошибки синхронизации как-то детектит или вы их можете отдетектить с помощью ошибок extension'a и закрыть соединение? Если вы не можете насищенно закрыть соединение, такой extension использовать не стоит ни в каком случае вообще.
- 4) Рассчитан ли протокол на persistent connect?
- 5) Если не рассчитан, используете ли вы какие-то дополнительные методики, чтобы подстелить себе соломы и ночью спать спокойно?
- 6) Если ваш протокол stateful, правильно ли вы с ним работаете, то есть у вас всегда один state, и вы этим state'ом, по сути, не пользуетесь? Либо вы резетите соединение, либо еще как-то искусственно проверяете, типа: if a!=1 then set a=1.
- 7) Используете ли вы connection pooling для экономии ресурсов?
- 8) Самое главное — готовы ли вы писать свои велосипеды? Мы написали два велосипеда.

hl++ HighLoad++

Спасибо!

Вопросы?

Контакты:

http://twitter.com/rvba_xek
s@ayerin.ru
<http://ayerin.ru/slides/>

hl++ HighLoad++

Badoo ищет классных людей!

Нам нужны:

- умный и вменяемый PHP/MySQL программист
- релиз-менеджер
- QA специалисты

Что даём:

- интересную работу
- 2000 евро за успешную рекомендацию

```
$> cd pub  
$> more beer
```

Нам нужны еще вменяемый PHP/MySQL программист, релиз-менеджер, QA-специалисты. Интересная работа у нас есть и 2000 евро. Подходите, я поболтаю с удовольствием за рекомендацию.

Вопрос из зала: За 2000 евро?

Сергей Аверин: Да. Теперь вопросы.

Вопросы и Ответы

Вопрос из зала: Вопрос номер раз. Было ли такое, что где-то вы решили сделать rconnect, внедрить его, но вы в результате отказались от этой идеи, попробовав. То есть что-то вы там не научились готовить или еще что-то.

Номер два. Было ли такое, что вы rconnect сделали и не получили, собственно говоря, никакого выигрыша, который ожидали?

Сергей Аверин: Мы всегда точно знали, куда стреляли. Того, чтобы сделали — и не получили выигрыша, не было. Это правда. У меня такая специфика, что у приложения очень маленькие запросы, они быстро выполняются и там время на соединение ко всем backend'ам — MySQL, поиск и прочее — достаточно весомо, сравнимо со временем запроса. Раза в 2 я всегда получаю прирост.

Теперь отвечаю на первый вопрос по поводу «сделали». Один раз была ситуация, когда мы стали использовать extension, который у нас in-house в компании разрабатывается для memcache. Он, я так понимаю, немного основан на коде оригинального, еще старого extension'a, который memcache для PHP. С ним была проблема, что возникали проблемы сбивки синхронизации, и они длились полосами пролежки по 15 минут. Я не мог закрыть соединение и сделать так, чтобы соединение заново открылось. С этим была проблема. Там соединения почему-то не закрывались.

Вопрос из зала: Решили проблему-то?

Сергей Аверин: Решили. Выкинули extension, написали свой, который умеет делать только 2 команды (set и get), но умеет детектить эти сбивки.

Вопрос из зала: В результате когда делали где-то, внедряли rconnect — везде получили выигрыши и везде смогли решить все проблемы с поддержкой протоколов и так далее.

Сергей Аверин: Да. Везде смогли решить все проблемы.

Вопрос из зала: Круто! Молодец.

(Смех в зале).

Вопрос из зала: Вопрос про MySQL Proxy. Вы реально его используете?

Сергей Аверин: Поскольку я говорю, что я не хожу в MySQL особо, у меня очень редкие запросы, я, в основном, хожу через handlersocket'ы. Даже в компании — нет, мы не пробовали. Я бы, наверное, хотел, подумал бы об этом.

Вопрос из зала: Просто вы говорили, что при сколько-нибудь существенных нагрузках очень непредсказуемо себя ведет, валится и вообще не очень хороший.

Сергей Аверин: Не могу ничего сказать. Врать не буду. Желание есть.

Вопрос из зала: Ты очень красиво рассказывал про самописные библиотеки с большим количеством соломы, но ты забыл вставить ссылки на GitHub.

Сергей Аверин: Блин. На самом деле, про эти библиотеки я рассказывал еще в прошлом докладе, когда это было на РИТе. Тем людям, кто мне писал в «Skype», я их с удовольствием давал. Короче, мне стыдно. Давайте договоримся, что я в адрес, где слайды, выложу до конца дня эти библиотеки, которые работают с handlersocket'ом, работают с memcache в виде PHP'шных файлов.

Вопрос из зала: Не сталкивался ли с протоколом Postres и использованием синхронизации?

Сергей Аверин: К сожалению, за 7 лет программирования не сталкивался с Postgres ни разу. Что сказать? Извините.

Вопрос из зала: Интересно, видел ли где-нибудь красивое API для асинхронного общения с базой? Когда ты шлешь пачку запросов одну за другой.

Сергей Аверин: MessagePack-RPC посмотри — очень интересный проект. Это, по сути, Binary JSON, к которому идут дополнительные плюшки в виде своего RPC-протокола, в виде готовых написанных библиотек. Он не предназначен для баз или не баз. Это протокол, очень сильно заточенный на сжатие коротких данных (именно коротких). Там почти на каждый формат данных есть два варианта.

Допустим, integer, которых больше 20-ти тысяч, и integer меньше. Integer меньше сжимается в 1 байт, а integer больше — в 2 байта или в 4. Там весь смысл в run length encoding. Там на уровне бит все работает. Очень интересный проект, сжимает очень круто и сильнее, чем Google Protobuf. В Google Protobuf'е на уровне байт протокол, а в MessagePack'е — на уровне бит.

Там есть Parallel Pipelining, там есть синхронизация запросов-ответов. В принципе, там это может быть асинхронно. Но только в том случае если ты хранишь некий пул посланных запросов. У тебя есть какой-то массив, в котором ты можешь сопоставить, кому пришел ответ. Это библиотека плюс протокол плюс протокол упаковки.

Вопрос из зала: Если использовать rconnect в PHP с теми же базами данных, это реально без каких-то внешних tools или переписывания extension'a?

Сергей Аверин: С чем? С `rconnect` в PHP с чем?

Вопрос из зала: С MySQL. `Pconnect` вместо просто `connect`.

Сергей Аверин: Можно использовать. Но дело в том, что там сделано очень хитро. Там тоже `run length encoding`. Там сначала дается длина команды, потом сама команда. Но жопа в том, что есть номер кадра, но он с каждым `request'ом` резетится. Ты послал запрос, у него приходит ответ с № 0, потом с № 1, потом с № 3... Эти ряды пакуются, по-моему, до 25-ти килобайт, что ли, эти кадры могут быть. Но когда ты посылаешь следующий запрос, ты опять получаешь ответ 0, 1, 2, 3.

Ты можешь использовать MySQL с `rconnect'ом`, если тебе это нужно. Для этого нужно использовать extension MySQLi, ставить вот это "р:" в начале хоста, и он тебе будет это соединение реинициализировать, каждый раз авторизовать тебя с логином и паролем заново. Этот момент и будет синхронизацией. Если в ответ на синхронизацию приходит какая-то фигня, ответ от предыдущего соединения, то у тебя будет ошибка, типа «мы пытались авторизоваться, а пришла какая-то фигня». Ты должен ловить эту ошибку, закрывать `connect`, открывать заново и работать с ним заново. Все, тогда у тебя будет все хорошо.

Во всей серверной логике, которая у меня ходит куда-либо, всегда 3-4-5 попыток делается. Защито, что если будет проблема, мы будем пытаться повторять. Причем на какие-то попытки коннект закрывается, на какие-то — нет. В зависимости от того, это ошибка синтаксиса, синхронизации, или это просто не найден ключ, тайм-аут произошел, или еще что-то случилось.

Вопрос из зала: Спасибо.

Сергей Аверин: Можно. Но надо делать повторы соединений и надо ловить ошибки и при этих ошибках закрывать его.

Вопрос из зала: Пробовал ли ты `rconnect'ы` с Redis'ом?

Сергей Аверин: Не пробовал. Когда я внедрял Redis, мне это было еще не очень интересно. Это был второй этап проекта, уже ускорение. Redis еще на уровне развития, взлета был выкинут, к сожалению, по причине того, что мы не научились его варить.

Вопрос из зала: На РИТ, помнится, рассказывали, что на PHP, в основном, пишете. Python'овская алхимия откуда? Личное увлечение?

Сергей Аверин: Просто, да. Личное увлечение, закончившееся покупкой лицензии на PyCharm, за что мне стыдно.

(Смех в зале).

Вопрос из зала: Солидарен. Спасибо.

Вопрос из зала: Вопрос немного не по теме: MongoDB используется?

Сергей Аверин: Подойди потом, я тебе покажу человека, который использует у нас. (Смех в зале). Он тебе все расскажет. Ну, а что, я врать, что ли буду? У нас внедрена MongoDB в одном из проектов внутри компании. Я тебя познакомлю с тем, кто его использует. Он тебе расскажет про слезы и печаль. (Смех в зале). Ну, или наоборот. Я обычно про слезы и печаль рассказываю. Меня прет.

Низкоуровневая оптимизация С/С++

Андрей Аксенов

hl⁺⁺ **HighLoad⁺⁺**

Низкоуровневые оптимизации

Андрей Аксенов, Sphinx
Highload⁺⁺2011

hl⁺⁺ **HighLoad⁺⁺**

Про что доклад

Андрей Аксенов: Доклад будет про «сантехников» интернета, так сказать, рыцарей говна и пара, которые ковыряются в потрохах. Точнее, нет, не про сантехников самих по себе. Я про них ничего не знаю. Про говно и пар будем доклад. Соответственно, чтобы вы могли, немного ознакомившись с ситуацией, принять осознанное решение, как ковыряться в говне или стравливать пар. Либо научиться это делать так, чтобы не сильно пачкаться, не ошпаривало и так далее.

hl⁺⁺ **HighLoad⁺⁺**

Про что доклад

- Высокие нагрузки => надо быстро
- Надо быстро => надо оптимизировать
- Надо оптимизировать => и так, и эдак
- Про высокий уровень говорить затрудняюсь
 - С примерами плохо, без примеров плохо
- Поговорю про низкий уровень

Низкоуровневая оптимизация. Почему доклад приняли на «HighLoad» я, честно говоря, не знаю. Но теория такая, что если высокие нагрузки, то надо быстро, значит, надо оптимизировать. Оптимизация, как известно, это, как преждевременная эякуляция, зло. Поэтому ею никто не занимается. Хотя, на самом деле, заниматься надо на двух разных уровнях. На высоком уровне — алгоритмическая оптимизация, правильные структуры данных, правильные алгоритмы, которые по ним бегают. Низкоуровневая — когда вы уже сделали свой хешик из строчек, то его можно херово реализовать (в смысле, как обычно), а можно правильно — и он будет работать хорошо.

Про высокий уровень я пока предметно говорить затрудняюсь. Для того чтобы говорить предметно, нужен конкретный пример — какой-нибудь безумный алгоритм, который никому на хрень не интересен, кроме лично меня. У меня он ровно в одном месте в проекте появился для решения какой-то сильно своеобразной задачи, и больше никому не нужен.

Можно поговорить про низкий уровень, потому что это «говно и пар». Оно возникает везде.

hi HighLoad++

Низкий уровень?

- Это, конечно, ассемблер
 - И только ассемблер
 - Иначе быть не может
 - Однако!

Низкий уровень в природе одни — это ассемблер, потому что иначе быть не может. Все, что выше ассемблера — оно пффф... Недостаточно низко. Однако!



Поскольку сейчас 2011-й год, и наноботы бороздят просторы «Большого театра»... Про наноботов мы беседовать не будем, но ассемблеров развелось много.

hi HighLoad++

Низкий уровень

- Однако “ассемблеров” в 2011 году много
 - C99, C++03, C++11, Java, C#, LLVM
- Поговорим, выходит, про C/C++
 - Несмотря, что

```
auto f = [](){}; // this is valid fucking C++ now
```

C99 — есть такой ассемблер. C++ в стандарте, ремикс 2003-го года, 2011-го и так далее. В принципе, GVM'ы и dotNet'овская виртуальная машина (да и Erlang'овская, в том числе) в каком-то роде тоже являются высокоуровневым ассемблером. Но я про них ни хрена не знаю, поэтому говорить будем про C++.

Почему C/C++

- Все еще ближе всего к железу
- Все еще топовые компиляторы
- Все еще НОД, интегрируется везде
- Все еще много стороннего софта на
- Ну и я других “ассемблеров” “не знаю”
 - “Ну то есть что считать за секс”

Почему про него? Достаточно близко к железу. Можно выпилить код, который понятно каким образом транслируется в машкод. Все еще компиляторы достаточно хорошо оптимизят, хотя всякие демоны, HotSpot'овские, Java'вские, dotNet'овские компиляторы поджимают. Это все еще наибольший общий делитель. С-компилятор есть везде, а вот какой-нибудь Erlang или долбаный Haskell поди задеплой, особенно на мобильном телефоне...

Все еще немало стороннего софта, который на этом самом С (либо, не дай Бог, C++) написан. Иногда приходится брать инструмент и нырять туда, в систему труб, сочленений и кранов.

Немаловажная деталь — я других не знаю ассемблеров. В смысле, надо понимать правильно. Код на какой-нибудь Java напишу получше некоторых, но пффф... Как устроены JVM'ы, как оптимизить — хер его знает. А это вот знаю.

Про... лозунги

- RT3D == Batch, batch, batch (c) ATI/Nvidia
- VLDB == Shard, shard, shard (c) everyone
- Multicore == Thread, thread, thread (c) Intel
- TDD == Test, test, test (c) K.Beck?
- Agile == Sprint, sprint, sprint (c) M.Fowler?
- Optimizations == ???

Про лозунги. Самое главное что? Лозунг. Потому что 90% уйдет с одной мыслью в голове, и надо, чтобы эта мысль была правильная. В разных областях человеческой жизнедеятельности есть разные лозунги.

Например, в RealTime 3D компьютерной графике лозунг, который принадлежит компаниям Nvidia и ATI — «Batch, batch, batch». В смысле, в секунду можно пропихнуть, грубо говоря, 500, 1000, 2000 запросов на отрисовку треугольничков, поэтому «Batch, batch, batch». Когда у вас большая база, и она ни хера не влезает в один сервер, то появляется немедленно слоган «Shard, shard, shard». Intel трудится, что есть сил, печатает «вафли», и на «вафлях» все больше и больше ядер — «Thread, thread, thread». И так далее.

Самая важная задача доклада была — придумать правильный слоган про оптимизации. Те 90%, которые уйдут с одной мыслью, должны уйти с правильной мыслью.



Лес рук. Что это будет за мысль? Нам срочно надо придумать лозунг, вам с ним жить.

Голос из зала: Fast, fast, fast.

Андрей Аксенов: Нет, «Fast, fast, fast» — это пикап какой-то.

Голос из зала: (Неразборчиво, без микрофона).

Андрей Аксенов: Это профессия такая. Опять же — сантехника, говно и пар.

Голос из зала: Там, скорее, важно (неразборчиво, без микрофона).

Андрей Аксенов: Тебя куда-то в глубокую философию понесло. Слоган должен быть резким, как удар кирпичом. Какие-нибудь идеи?

Голос из зала: (Неразборчиво, без микрофона).

Андрей Аксенов: Это вырезано цензурой. (*Смех в зале*). Хорошо.



Лучшее, что я сумел придумать (достаточно плохое при этом), это слоган «Bench, bench, bench». Это нематерное слово, насколько я знаю. В смысле, «benchmark'ать, benchmark'ать и еще раз benchmark'ать», как завещал великий Ленин. Толку (если неграмотно делать) от этого не будет, но вреда тоже. Я не медик, но клятва Гиппократа соблюдена.



256 оттенков серого

- Но нас интересуют только 3
- В целом – диск, память, CPU
- Сегодня – память, CPU

- Завтра – Ulrich Drepper, Agner Fog
 - Звоните, все трое и звоните

Ближе к телу. Тел (в смысле, bottleneck'ов) в природе бывает 3 штуки. Можно упереться в диск, можно в память, можно в CPU. Сегодня нас интересуют из них только два, поскольку с диском и так понятно. Пробежимся по верхам. Кому интересны не верха — есть основополагающие труды товарища Drepper'a про память и товарища Agner'a Fog'a про CPU. Все 3 человека, собираются их читать, в принципе, могут либо нагуглить, либо просто мне позвонить, и я sms'кой отправлю ссылку.



Что нужно знать про RAM

- Как соотносятся разные цены доступа
- Что память работает не байтами...
- Что бывает L1/L2 кеш
- Что кеш вымывается
- Что (не)выравнивание небесплатно
- Что бывает (и иногда работает) префетч

Что надо знать про память на базовом уровне на трех пальцах за 3 минуты на коленке. Что ценник доступа очень разный — в зависимости от того, как вы эту память дрочите. Что память — это не какая-то волшебная штука, у которой с константной ценой доступ к каждому долбаному биту, а несколько более сложная штука.

Что она работает не байтами, прежде всего, а линейками байт того или иного размера. Что у памяти бывает — ха-ха! — L1, L2 кеш, хотя он бывает не в памяти, а на проце. Тем не менее, к несчастью, он бывает.

Что этот кэш можно вымыть, смыть в унитаз. Тема такая.

Что с памятью лучше работать по выровненным адресам, чем по невыровненным. Повторюсь — оно неравномерно. Доступ к байту по смещению, не являющемуся степенью «двойки» (условно говоря), оказывается медленнее.

На закуску, что называется, на оценку «5+», что всякие фокусы с ней можно делать и прочие префетчи. Они иногда помогают.

hi++ HighLoad++

Что нужно знать про CPU

- Что есть регистры, инструкции, кеш кода
- Что есть целая арифметика, FPU, MMX, SSE
- Что есть *mu-ops*, pipes, branch prediction, register renaming, out-of-order execution
- Что все эти чудеса стоят разных “денег”
- Что amd64 “лучше” i686, когда не “хуже”

Про CPU то же самое. Краткий опорный план, что нужно понимать, когда ты собрался нырнуть в оптимизации и что-то в оптимизациях накопать на низком уровне. Что есть регистры, инструкции — как, в принципе, устроен проц. С регистрами, инструкциями, кешом кода. Что стоит какая инструкция, какие есть классы инструкций. Как можно посчитать одно, другое, третье. Как со всем этим жить.

hi++ HighLoad++

RAM, про цены в целом

- Bandwidth. Тупое линейное чтение
 - 4.6 GB/sec на ноутбуке, 12.7 GB/sec на сервере
 - Ништяк, ништяк?
- Latency. Такие же 100M int32, stride 4096
 - 195 MB/sec, 2.14 sec/100M, ~49 Mreads/sec
 - Т.е. последовательный доступ ~1-2 такта
 - Т.е. случайный доступ ~40-60 тактов

Более подробно. Глава № 1 про ту самую память. Чего стоит память, грубо говоря. Чего стоит работа с памятью.

Два стандартных показателя — Bandwidth (полоса пропускания) и Latency. Тупое линейное чтение из памяти это Bandwidth. Получается 4 гигабайта в секунду на ноутбуке, 12 с чем-то на сервере. Примитивный benchmark, который честно читает память, честно суммирует регион памяти и потом печатает все дело на stdout, чтобы компилятор, сволочь, не соптимизировал и не выкинул, и не получилось время 0 и скорость бесконечна.

Казалось бы, все прекрасно. Скорость каких-то бешеных гигабайтов в секунду. Ура-ура, ништяк-ништяк.

Ни хера подобного. Как был устроен benchmark, который получил 4 гига в секунду? Брал 100 миллионов 32-битных чисел и втупую их суммировал. Линейно. Нулевое, первое, второе, третье. Линейно бежал по памяти.

Теперь делаем следующий финг ушами. Берем и суммируем число № 0, число № 1024, число № 2048 и так далее. Потом точно так же с шагом в 1000 int'ов, начиная с первого. Потом начиная со второго. Так 1024 раза.

Вместо того чтобы линейно все просуммировать, просуммировали с дикими прыжками по 1тысяче int'ов по 4 килобайта и повторили этот эксперимент 4 тысячи раз. На кой хрень? Для того чтобы симулировать получаемый доступ к памяти, который плохо кэшируется.

В основном, на ноутбуке все benchmark'уют, поэтому цифры про него, где не указано обратное. Внезапно вместо 4-х гигабайт в секунду получается 195. 2 секунды на 100 миллионов случайных доступов к памяти, ля-ля-ля, 50 миллионов в секунду. В пересчете на такты процессора, которых тоже заливают ограниченное количество под крышку черепа, последовательный доступ выливается в 1-2 такта процессора, что почти идеально — лучше быть не может. В пол-такта сложно вообще уложиться. Случайный доступ существенно дороже — в десятки раз. 40-60 тактов.

Немаловажная деталь. Случайным образом скакать по памяти дорого. Причем настолько дорого. Вместо пяти гигов в секунду внезапно и полгига не получается.

hl HighLoad++**

RAM, про цены на практике

- Читать один поток надо мало... но редко
- $100M \times \{ *c++ = (*a++) + (*b++); \}$
- Линейно **1111 MB/s** (вместо 4400+)
- С шагом 4KB выходит **36 MB/s** (вместо 195)
 - Падение в 5.4 раза
 - Жалких 9 Mop/s, 240 тактов на одну (omfg)

Дальше ситуация ухудшается, потому что на практике вам почти никогда не надо будет читать один поток данных из памяти и что-то с ним делать. Скорее всего, вам надо будет делать с вашими данными что-то более противоестественное.

Например, второй тривиальный benchmark. Давайте возьмем два шматка памяти и просто сложим. Два вектора сложим в памяти. Больших вектора, по 100 миллионов чисел. 100 миллионов раз проделаем операцию $*c++ = (*a++) + (*b++)$, где c , a и b считаются, соответственно, либо линейно, либо из случайных мест памяти. Линейно падение скорости составляет 4 раза. Мы видим, что помимо собственно чтения и упора в память, цикл увеличения указателей и все эти радости влияют, потому что он настолько быстр. Тем не менее, скорость падает более ожидаемых трех раз.

Если все это делать псевдослучайным образом (случайно скакать по памяти с большим шагом, чтобы симулировать постоянное выбивание кеша), то получается совсем тоскливо. Такое дело дает 36 мегабайт в секунду и на одну операцию сложения двух долбаных чисел, которая, формально говоря, выполняется за один такт процессора, мы тратим не 1 такт, а 240 примерно.

hl HighLoad++**

RAM, L1 кеш

- Почему так? Потому что L1/L2 cache
- Скачаем с шагом N => кеш-миссы => тормоза
- Шаг 4..64, ~4400..330 MB/sec, ~2x/шаг
- Шаг 64..1024, ~330..195 MB/sec
- Значит, размер L1 кеш-линии 64 байта :)

Почему так все плохо? Потому что память — повторюсь — это не магический черный ящик, который просто работает быстро, как пулемет. Несколько более сложная структура и медленная вдобавок ко всему. Чтение из памяти проходит через каскад тампонов (через 2-3 тампона).

Напрямую с памятью процессор никогда не работает, по большому счету. Вместо этого он сначала идет в кеш первого уровня, который на самом кристалле маxонький (8-16 килобайт), пытается в нем что-то найти. Если неудача — идет в более большой кэш второго уровня, который опять на кристалле. Если опять неудача — идет собственно в основную память и, поскольку это сравнительно медленно, подтягивает из этой самой основной памяти не 1 дополнительный байт, который вы запросили, а полоску — 32, 64, 128 байт.

Соответственно, ровно по этой причине вот так все и работает, когда мы, чтобы вынуть очередное число из буфера, скакаем с шагом 4 килобайта. На каждые 4 байта внезапно кеш-мисс, подтягивание 64-х байт из памяти — и все плохо.

Вот такой незатейливый *benchmark*, который меняет шаг, он это крайне наглядно показывает. Читали линейно — у нас практически не было лишних кеш-миссов. Увеличили шаг в 2 раза — каждые 64 байта как читались за 1 удар, так и читаются. Но теперь этих линеек по 64 байта мы считаем ровно в 2 раза больше, потому что на первом проходе читаем каждый четный *int*, на втором — каждый нечетный. Скорость падает ровно в 2 раза и падает. Соответственно, вплоть до шага 64 байта скорость линейного чтения из памяти падает, как и предсказано наукой, по 2 раза на каждое увеличение шага в 2 раза. После этого стабилизируется и начинает падать медленнее. Значит, размер L1 кеш-линии на том процессоре, на котором я тестировался, 64 байта.

HighLoad++

RAM, L2 кеш

- А давайте иначе:
вы мне — свою душу,
а я вам — домашний
адрес епископа
Диомида?

Изменяя эксперимент. Меряем другую штуку. Только что обмерили L1 кэш и увидели, какой он эффект дает. Если никуда не скакать, если читать по 4 байта с шагом 4 байта, то 4400 мегов секунды. Если 4 байта читать каждые 8 байт — 2200, 4 байта каждые 16 — 1100 и так далее.

Изменяя эксперимент, производим обмен в обратную сторону.

HighLoad++

RAM, L2 кеш

- Фиксируем шаг 1024, уменьшаем **данные**
- 100M, ..., 4M, 3M, 2.3M == 195 MB/sec
- **2M == 648 MB/sec**
- **1M == 1688 MB/sec**
- **512K == 1724 MB/sec**
- Все сходится, на лаптопе 2MB L2 cache

Вместо того чтобы менять шаг, меняем данные. Обрабатываем не полектара данных, а шматок поменьше. Поведение вот такое. До тех пор пока буфер уменьшается вплоть до определенного предела — до 4-х, 3-х, 2,3 мегабайт (не спрашивайте, почему именно 0,3 — я не знаю). На 2,1 уже начались краевые эффекты.

Скорость одинаковая — 195 мегов в секунду (чтение). Тормоза. Начиная с 2-х мегов размера буфера, который многократно суммируется, начинает ускоряться. При размере буфера 1 мег и сотни итераций — еще ускоряться и так далее.

Опять же все сходится. Почему так произошло? Почему в тот момент, когда мы суммируем буфер 2 мега, прыгая по нему шагом, у нас скорость 600 мег в секунду, а как только буфер 2,3 мега (заметьте) — скорость 195 мег в секунду (внезапно в 3 с чем-то раза меньше). Потому что все сходится. На лаптопе 2 мега кеша, а не 2,3. Эти лишние 0,3 оказывают именно такой смертельный эффект.

HighLoad++

RAM, про выравнивание

- Лаптоп, 4.6 vs 4.2 G/sec, минус 8.3%
- Сервер, 12.7 vs 11.6 G/sec, минус 7.3%
- Intel, SPARC, ARM

Последняя мелкая деталь. Про то самое выравнивание. Проводим третий эксперимент. Вместо того чтобы читать по тому адресу, который операционка отдала, который выровнен, потому что любой вменяемый аллокатор данные выравнивает, прибавляем к нему «единичку». Совершенно внезапно скорость падает на 7-8% просто за счет того, что ты читаешь данные не по выровненному на границу 32-х бит адресу. Это, на самом деле хорошая ситуация, которая на архитектуре Intel. Оно, во-первых, работает, во-вторых, тормозит всего на 8%. На SPARC'е оно просто на хер падает, а на ARM'е оно работает, но читает не тот результат.

HighLoad++

RAM, выводы

Какая мораль всей этой длинной и непонятной басни. Выводы. Картинка — для привлечения внимания.

HighLoad++

RAM, таки выводы

- Если данных много, лучше линейно
- Если линейно никак, лучше мало данных!
- Если можно переложить Тетрис, то нужно
- Если можно, лучше всякое выровнять
 - Но держать баланс с локальностью!

Выводы достаточно простые из всех затейливых benchmark'ов (на самом деле, крайне незатейливых, и тем не менее).

Если вам надо обработать очень много данных, намного лучше делать это линейно. Максимально линейно, насколько возможно.

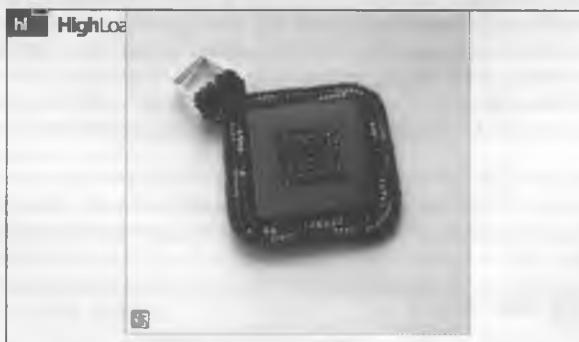
Если линейно никак не получается (а зачастую ни фига не получается, естественно), то нужно всеми усилиями добиваться того, чтобы рабочий working set влезал хотя бы в L2 кэш (2 мега, 4, 12 — сколько у вас на целевой железке).

Если можно поиграть в Тетрис и переложить данные в памяти так, чтобы они лежали более локально, чтобы работа могла производиться как раз без произвольных скачков по памяти, а хотя бы с помощью произвольных скачков в пределах L2 кэша (2 мега), то обязательно нужно поиграть в Тетрис и это сделать. Потому что — еще раз подчеркиваю. 2,3 мега — 200 мег в секунду обработка, 2 мега — 700 мег в секунду. С Божьей помощью, уменьшив размер данных и переложив их так, чтобы обрабатывать по 1,1 элемента за раз, увеличили скорость теоретически в 3 раза. В Тетрис играть — надо.

Ну, и почти бесплатный перк. Если можно это сделать, то и чтения и записи нужно выровнять на границу скольких-то байт (естественно, если это не мешает всему остальному). Профит от этого дела — 7-8%. От непопадания в кеш можно просрать значительно больше.



Начинаем с чистого листа. Глава № 2, почти не связанная.



Про процессор и всю ту дрянь, которая происходит от процессора, про которую хочется знать.

CPU, про регистры

- i286 – ax, bx, cx, dx, si, di, bp, sp
- i386 – eax, ebx, ecx, edx, esi, edi, ebp, esp
- i686 – eax, ebx, ecx, edx, esi, edi, ebp, esp
- amd64 – rax..rsp, r0..r31
- fpu (aka i287) – fp(0)..fp(7)
- sse – xmm0..xmm7

«Минута истории». Когда-то давным-давно, 30 лет назад был процессор 8088, у которого были 8-битные регистры al, bl и другие. Потом придумали 286-й. У него 16-битные регистры, совместимые с... Потом 386-й — у него 32-битные регистры, совместимые с.... Потом, потом, потом... Так вся эта дрянь до сих пор и тянется.

В любом современном процессоре у вас найдется набор регистров, который уходит по причинам совместимости где-то лет на 30-40 назад и восходит к родным i88 или что-то типа того. «Минута истории» на канале «HighLoad».

Важно то, что процессор — это такая машинка, у которой есть какой-то набор регистров. Тут, скорее всего, может быть какая-то фактическая ошибка, потому что я хотел проверить, сколько конкретно новых клевых 64-битных регистров на amd64 архитектуре. Вроде бы 32, но не помню, а в «Википедию» слазить не смог, потому что инет упал.

Есть набор регистров, с которыми, собственно, машинка работает (тот самый процессор). До тех пор пока вы в эти регистры укладываетесь, все хорошо. После того как вы в эти регистры влезть перестали, все у вас плохо. Отсюда — внимание — мгновенный и очевидный вывод. Почему архитектура amd64 (64-битные сервера) «намного круче» (в кавычках), чем, например, архитектура 686 или 386-я, с которой она, в принципе, совместима?

Просто потому, что у вас для проведения сложной счетной операции доступно намного больше регистров в любой конкретный момент времени. На самом деле, кажется, что их на i386 много — 8 штук. Но как только тебе надо сложить 3 числа и как только надо держать 3 указателя на 3 разных буфера, собственно 3 этих числа, какой-то счетчик в голове, у тебя мгновенно регистры кончатся. Тебе постоянно надо дергать память, сходить за данными в память, а память — это медленно. В тот момент, когда L1 кеш, это еще не очень медленно, но можно прилипнуть.

Поэтому 64-битная архитектура (amd64) намного лучше. Регистров больше, соответственно, влезает всякого счетного в них больше. Получается в целом быстрее. Она лучше, конечно, ровно до тех пор, пока не хуже, потому что от перехода там тоже могут всякие спецэффекты произойти.

CPU, про инструкции

- this->a += b
 - mov eax, [esi+12] ; The Load
 - add eax, ebx ; The Hit
 - mov [esi+12], eax ; The Store
- a += b
 - add eax, ebx ; The Hit

Про те самые инструкции. Внимание, C++. Доклад, который начинался про C++. Первый кусок кода, который в нем появился — про ассемблер. Как на самом деле работает инструкция что-нибудь типа «давайте возьмем и прибавим к члену класса что-нибудь». Она не работает в одну волшебную инструкцию «давайте возьмем и прибавим к члену класса что-нибудь». Она разбивается на микродействия, если угодно — инструкции. Примерно такие 3.

Сначала подтягиваем значение, с которым собираемся работать, из памяти в регистр. Увеличиваем это значение, вдобавок в предположении, что наша переменная *b* уже лежит в регистре. Кладем ее обратно. Сравните с кодом, который теоретически может сгенерить компилятор на выражение *a+b*. Он ровно втрое меньше. Невероятно, но факт.

HighLoad++

CPU, VS 2005 vs gcc 4.x

- this->a vs a
- member write-pressure


```
int iRes = m_iCounter; // skip nonwhitespace
while ( m_iCounter<m_iLimit &&
       m_sBuffer[m_iCounter] )
    m_iCounter++;

```
- test1 = 701 msec, test2 = 413 msec

Несмотря на то что космические корабли уже бороздят просторы, компиляторы не все и не всегда с такой херней справляются. Вот такой примитивный цикл в 2 строки, который активно использует какой-то счетчик в классе, крайне плохо оптимизируется с компилятором VS 2005 (и, по-моему, с VS 2008 тоже). Под рукой не было, проверить не удалось.

Тест № 1. Вот этот код как он есть. Сколько-то много итераций — 1 миллион, 100 миллионов, 1 миллиард, что-то вроде того. 700 миллисекунд.

Тест № 2. Берем переменную Counter и вместо нее везде пишем вот этот самый iRes. В самом конце записываем iRes обратно в поле класса Counter. Совершенно внезапно код, сгенерированный «Visual»'кой, начинает работать за 400 миллисекунд, а не за 700. Разгоняется в 2 раза.

Что в плюс товарищу gcc. Gcc такой дурью не занимается. Он видит, что можно в течение цикла работать с регистром. Он эту запись догадывается вынести за регистр сам. Visual'ка, соответственно, не догадывается.

Соответственно, в «Studio» можно поймать такую смешную оптимизацию. Если вы достаточно активно используете член класса, то иногда компилятор с этим не справляется. В данном случае «косяк» компилятора, который в «Visual Studio». Но они все грешные. Это не значит, что gcc идеален. Это значит, что gcc лажанется в каком-то другом месте, более увлекательном, про которое мы еще не знаем.

Как понять, что происходит, поймать такой баг. Не знаю, как. Надо знать, что процессор работает с инструкциями и примерно прикидывать, что какая делает. Глянуть в дизасс, схватиться за голову, что там такой фарш из трех инструкций, куча совершенно ненужных загрузок из памяти и записи в память, ужаснуться и попытаться что-то сделать.

В данном случае решение (повторюсь) — временно вынести член в локальную переменную. Внезапно код разгоняется в 2 раза синтетически.

Hi HighLoad++

CPU, про кеш кода

- Код это тоже данные и тоже память...
- Внезапно, функции и конвенции вызова
- Внезапно, инлайнить или нет?
 - Функции, `inline`, `__forceinline`, итп мычки
 - Классы, реализация в декларации
 - Шаблоны, функторы против коллбэков

Еще один замечательный момент, который надо понимать про CPU. Опять же все небесплатно. Код — это тоже данные, которые тоже лежат в памяти. С ним тоже надо как-то работать, поэтому нельзя сделать длинную простыню кода, которая чудесным образом будет быстро читаться. Внезапно возникает целый ряд увлекательных проблем. Во-первых, вместо того чтобы весь код подставлять вместо вызова, надо зачем-то вводить функции, чтобы размер бинарника был 1 мегабайт, а не 200.

В некоторых местах вместо того чтобы плодить вызов функции, надо функцию инлайнить. Внезапно возникает понятие конвенции вызова. Дескать, если мы уж делаем вызов функции, то как именно мы это делаем? Как именно мы передаем аргументы, как именно мы собираемся возвращать значение?

В общем, из-за того, что код — это тоже данные, тоже лежат в памяти, доступ к ним тоже не бесплатен, внезапно возникает целый ряд замечательных, глупых вопросов. Что инлайнить, что не инлайнить, как с этим быть.

Не совсем понятно, есть ли время останавливаться. Наверное, нет. Значит, тему инлайнинга, шаблонов, функторов и так далее, боюсь, придется заныкать в кулуары, замести под ковер и пытаться бежать дальше.

Hi HighLoad++

CPU, про креативное

- Pipes, начиная с i586 (Pentium-1)


```
add eax, ebx
      add ecx, edx      ; pairs
```
- Дальше ситуация ухудшилась!!!
- Mu-ops, renaming, out-of-order, и всякие другие увлекательные вду

Еще один креативный момент, который приходится знать про процессор. Современный процессор может выполнять инструкции в параллель. Начиная, по-моему, с Pentium-1 появились Pipes. Там их было 2, поэтому было очень прикольно писать код. Берешь и херачишь на ассемблере в 2 строчки. Если ты правильно эти строчки написал (а именно развел зависимости и 2 инструкции совершенно независимы друг от друга), то 2 инструкции спарились и выполнились за 1 такт.

Дальше ситуация ухудшилась, поскольку процы стали более сложными и шибко умными. Вместо двух явных конвейеров внутри их уже хрен знает сколько. Операции более сложные чем сложение, на самом деле, боятся на микрооперации. Регистры, поскольку их вечно не хватает, внутри себя процессор может как-то ловко переименовать, инструкции может хер знает

как перестроить (в смысле, выполнить не в том порядке, в котором они не то что в С'шной программе написаны, а даже не в том порядке, в котором они в машкоде написаны).

Происходит куча другого увлекательного буду. Здесь у меня понимание отказывает, если что. Я просто знаю, что оно есть, и знаю это для одной простой вещи, если угодно. Смотришь в машкод — видишь фигу. Не всегда то, что ты видишь даже в машкоде, коррелирует с тем, как устроена реальная жизнь.

HighLoad++

CPU, про ветвления

- if (a!=0) { Doit(); } ...

```
    cmp eax, 0
    jne Label1 ; branch point
    call Doit
Label1:
```

...

- likely(), unlikely() etc

Последний момент про важные с точки зрения оптимизации внутренности CPU. То, что есть понятие ветвления, несмотря на то что оператор GoTo много лет назад заклеймили, и его почти нигде нет на высоком уровне. Внутри в процессоре он прекрасно есть. Вот такой С'шный код в одну строчку транслируется в несколько инструкций. Сравниваем регистр с чем-нибудь, делаем условный переход.

В тот момент, когда происходит переход, это — барабанная дробь — опять не бесплатно. Да что ж такое — ни доступ к памяти не бесплатный, ни в процессоре ничего не бесплатнo. Он здесь не бесплатный, потому что переход условный. Процессор-то не знает, куда прыгать, куда бежать. Раньше заранее ничего подпредметить вообще не мог. Сейчас процессоры «умные», пытаются следить, где какой переход куда перешел и заранее подтягивать из памяти именно тот код, с которым следующей инструкции, вероятно, придется работать. На кой хрен это надо знать с точки зрения оптимизации на языке хоть чуть-чуть выше ассемблера? Первое и второе.

Первое. Ветвления опять же не бесплатные ни фига. Второе. В компиляторах энное время назад появились всякие увлекательные макросы, типа analysis likely, unlikely и так далее, которые позволяют вам хинтить компилятору, что этот переход в 99% случаев пойдет не туда. Теоретически это позволяет вам выпилить более быстрый код. Пример я сделал не успел.

HighLoad++

CPU, еще про ветвления

- switch() vs простыня простых if()
- Ни разу не эквивалентны!!!
- switch() оптимизируется, if() нет
- Эффект заметен от 3 (трех) значений

Последний момент про ветвления. Это уже мы поднимаемся, наконец, от уровня говна к уровню пара. Все, что было до этого, было про битики, байтики и микросхемы. RAM, CPU и так далее. Мы, наконец-то, начинаем всплывать на уровень C++.

Это чисто С'шный прикол, что называется. Давайте напишем 2 совершенно аналогичных кода. Первый – с простыней if'ов. if ($a==1$) ... else if ($a==2$) ... else ... и так далее. Или давайте напишем switch(). Что будет работать быстрее, внимание, вопрос.

Правильный ответ — switch(). Удивительный правильный ответ. Он мало того будет работать быстрее. Он будет работать быстрее, даже когда в этом switch()'е всего 2-3 значения. Какого хрена? Когда вы делаете тот самый switch() с набором константных значений, компилятор делает всяческое вду опять же. Считает, например, оптимальную хеш-функцию от входного аргумента, строит таблицу переходов. Опять же — занимается всячим увлекательным вду, но в итоге делает намного меньше работы.

Грубо говоря, один линейный расчет функций, который сравнительно быстрый, и один jmp. В отличие от простины if(), в случае которой компилятор обязан генерировать кучу инструкций «strcmp — перешли — strcmp — перешли — strcmp — перешли»... Переход ветвлений не бесплатный. Процессор шизеет. «Богородица, вернись».

HighLoad++

CPU, if-for vs for-if

- for (int i=0; i<NITER; i++)
 $res += (a==b)$
 $? c*d-e$
 $: (a+b)*c+d;$
- For-if, 0.274 sec, res 1 300 000 000
- If-for, 0.184 sec, res 1 300 000 000

Benchmark. Наконец-то, еще 3 строчки C++. Насколько это дело не бесплатное. Первый вариант кода — вот оригинал, в котором я для краткости вместо if'а написал адский тернарный оператор. 274 миллисекунды. А, б, с, д константы, на протяжении цикла, как видно, не изменяются. Тем не менее, компилятор до этого догадаться не может.

Если мы вынесем if $a=b$ за пределы цикла и напишем, соответственно, if $a==b$ один цикл, иначе другой цикл — в полтора раза. Понятное дело, что это полтора раза на крохотной синтетике. Понятное дело, что на общем фоне вы из подобных оптимизаций добьетесь максимум 1 — 10%. Но курочка по зернышку. К вопросу о сильной небесплатности ветвлений и что с ней делать.

HighLoad++

CPU, про FPU, SSE

- Дивный отдельный мир
- fpusum, 44042 usec, msvc default
- fpusum, 15386 usec, msvc /fp:fast
- ssesum, 8453 usec
 $_mm128 res = _mm_set_ps1(0.0f);$
 $while (p < pmax) res = _mm_add_ps(res, *p++);$

Еще есть отдельный дивный новый мир, который связан с вычислениями про плавающую точку. Мне явно надо ускоряться, поэтому очень поверхно.

В «Visual» FPU-часть очень тормозная по умолчанию. Удивительный факт № 1 — ее можно резко разогнать ключом командной строки. Внезапно начинает работать втрое быстрее.

Удивительный факт № 2. Написав достаточно простенький код... На слайде — код суммирования того самого массива из 100 миллионов с плавающей запятой, написанный на SSE. 2 строчки, которые страшные, но не смертельные. Он работает еще в 2 раза быстрее, как видите. Или в 6 раз быстрее, чем то оригинальное говно, которое генерит «Visual».

hi HighLoad++

CPU, про FPU, SSEx

- gcc 4.4.3, amd64
- fpusum, 11214 usec
- ssesum, 3414 usec

В случае с gcc тоже не все так хорошо. Несмотря на то что gcc в последнее время умный и на правильной платформе генерит вместо тормозного FPU-кода сравнительно быстрый SSE-код, он все равно его генерит по одному элементу за раз. Вручную выпиленный цикл суммирования чисел (синтетический пример, конечно), тем не менее, работает быстрее под gcc тоже.

Отдельный дивный мир про FPU, про который за 3 секунду можно сказать только то, что иногда он очень хорошо перекладывается и на SSE, векторизуется, потому что SSE — это минимум 4 флота за раз, в зависимости от версии, а в более толстых версиях — и побольше. Соответственно, работая с толстыми 128-битными регистрами по 4 флота за раз, удается выпилить до 3-4 (иногда и больше) раз по скорости. Отдельный перк от SSE еще такой, что некоторые странные инструкции выполняются существенно быстрее, чем на FPU. Например, корень квадратный считается на SSE с дикой скоростью, по сравнению с...

hi HighLoad++

CPU, SSE unroll

- ssesum, 8453 usec
- ssesum x4, 8240 usec
- ssesum x4 + non-naïve, 8056 usec, +5%

```
res = _mm_add_ps(res,
    _mm_add_ps(_mm_add_ps(p[0], p[1]),
    _mm_add_ps(p[2], p[3])));
p += 4;
```

В том же самом benchmark'е мне, наконец, удалось получить эффект от ручного разворачивания цикла. Годы идут, а приемы ни хрена не меняются. Как рукой развернуть цикл 100 лет назад давало эффект — так оно до сих пор дает. Другое дело, что эффект не сильно гигантский, тем не менее, свои 5% выпилить удается.

Что делают 2 магические строчки. По сравнению с примитивным суммированием по одному флоту за раз. Во-первых, мы переписали все на SSE и суммируем по 4 флота за раз. Во-вторых, вместо того чтобы при суммировании векторов постоянно дописывать что-то к аргументу, мы немного разнесли зависимости, что позволило процессору спарить инструкции лучше, и дало свои честные 5%.



Последняя часть самого главного врага во всех этих битвах с оптимизациями. Последняя теоретическая часть, точнее. Про компилятор.

Что нужно знать про cc/libc

- Компилятор == gcc, MSVC, (припадочно) ICC
- Про флаги сборки (arch, etc)
- Про конвенции вызова
- Про волшебные define-ы

Примерно что нужно знать про компилятор. Что используем в качестве компилятора — вроде понятно. Надо знать про всякие удивительные флаги сборки, которые иногда могут сильно удивить, про конвенции вызова, про волшебные define-ы (хотя это больше про MSVC).

cc, флаги сборки

- gcc, -march=(i386|i686|core2???)
— Умолчания системы могут удивить
- msvc, /arch:(sse|sse2), /fp:(fast|precise|...)
— По умолчанию голимый FPU
- msvc, /ZI (aka E&C, edit and continue)
— По умолчанию включен... "language", my ass

Флаги сборки. Если выставить не ту платформу, то компилятор сгенерит код из предыдущего столетия, который рассчитан на процессор 20-летней давности i386, и будет ни хрена не оптимально исполняться. Надо зорко озираться, потому что умолчания даже на Linux'овых системах могут дико удивить. Я регулярно вижу системы: вроде 64-битный сервер — генерит 32-битный код, который на 64-битной платформе исполняется достаточно хреново.

У «Visual» традиционно с этим нехорошо. Она традиционно дефолтится в i386, по большому счету. Ее надо руками подталкивать и ставить ей архитектуру более умную, чем 386. Руками ей подсказывать, что все-таки SSE уже наступил везде. Вдобавок надо делать какие-то фокусы, чтобы она чуть более внятно обрабатывала операции с плавающей точкой (потому что по умолчанию она какой-то совершенно безумный код генерирует).

Еще одна специфическая для «Visual». Это знаменитый edit and continue. Совершенно замечательная фича, ни разу не пользовался. Тем не менее, как-то раз случайно выяснилось, что когда у вас включен edit and continue в дебажном build'e, то дебажный build тормозит не просто как дебажный build, а еще в 3 раза.

За фигу, которой я никогда не пользуюсь, неплохая плата. К вопросу о том, что надо знать о компиляторе. У него есть настройки, и их надо крутить, иначе результаты удивляют.

hi HighLoad++

СС, конвенции вызова

- __cdecl, __stdcall, __fastcall
- intrinsics (msvc) aka builtin functions (gcc)
 - А это не магия! fabs(), strcpy(), итп
- _SECURE_SCL итп отладочные итераторы
- msvc 2005 по умолчанию...
 - __cdecl, NO intrinsics, _SECURE_SCL 1

Конвенции вызова — опять та же самая штука. По умолчанию могут удивить, да. Этим страшает, в основном, «Visual». У gcc ситуация получше, но тоже иногда накалываешься. По умолчанию он дефолтится в сравнительно тормозной __cdecl. Это конвенция вызова. Все данные, которые будут переданы в функцию, которую вы вызываете, гоняются через память, через стек. Соответственно, ставишь __fastcall — внезапно все начинает работать чуть быстрее. Intrinsics тоже по умолчанию в «Visual» отключены. В gcc, слава Богу, включены.

Intrinsics, несмотря на страшное название, это не какая-нибудь черная магия. Это функции типа взятия модуля, копирования строк и так далее. Суть в том, что когда тот же strcpy() можно заинлайнить, gcc это по умолчанию честно делает. «Visual» — нет. Ей надо опять же подпихивать рукой. Еще она очень любит включить отладочные итераторы (для тех, кто пользуется SCL), что опять сажает перфоманс в разы.

Краткое summary слайдов — с gcc, наверное, не очень надо зорко озираться. С «Visual» — глаз да глаз. По умолчанию она какие-то совершенно параноидальные настройки ставит. Наверное, они для безопасности такие, но код генерят тормозной.

hi HighLoad++

СС, аллокации

- Старые недобрые malloc()/free()
 - Drop-in замены: nedmalloc, tcmalloc
- Мало “больших” (4-16K+) везде ок
- Много мелких аллокаций везде боль
 - Например, 1 М аллокаций по 16 байт
- Ручные пулы все еще работают!

Совершенно не связанный с конкретным компилятором момент — это и беда 99% программ (если не 100%). Это борьба с аллокациями. Старая добрая аллокация памяти как 20 и 100 лет назад была медленная, так и до сих пор.

malloc()/free() — дорогое явление. Крайне желательно его хотя бы ликвидировать. Крайне желательно его заменять на какой-нибудь более внятный аллокатор. Делать большие алло-

кации памяти — на любом компиляторе в любой среде будет более или менее приемлемо. Много мелких аллокаций из коробки до сих пор никто не делает. Это до сих пор боль.

Несмотря на то что теоретически в библиотеках, в операционной системе и так далее поддержка для мелких аллокаций должна бы совершенствоваться, очередной раз (раз в три года) выпилив ручной пул на миллион объектов, в котором ты вручную распределяешь элементы, с удивлением выясняешь, что все равно это до сих пор быстрее, чем пользоваться `malloc()/free()`. Прогресс местами куда-то идет, но местами стоит на месте. Аллокации — это как раз такое место.

HighLoad++

Боевой пример

- Морфологический словарь, libaot
- Сторонний код, между прочим!
- Как именно удалось его взгреть в 3 раза
- Что делает тот критичный код?
 - возвращает набор лемм по словоформе
 - СТАЛИ -> СТАТЬ, СТАЛЬ

Боевой пример. Куча всяких несвязанных фокусов и фактов. Как они, собственно, применяются иногда на практике. Тоже, наверное, интересно. Может быть, и нет. Я не знаю.

Боевой пример вот какой. Морфологический словарь АОТ, секретный проект по вкручиванию этого самого морфологического словаря в Sphinx был выбран за то, что, во-первых, показательная оптимизация, во-вторых, с Божьей помощью, это сторонний код (не я его писал). Тем не менее, его удалось взгреть в 3 раза. Подчеркиваю — взгреть совершенно без каких-то алгоритмических фокусов. Алгоритм какой был, такой и остался. Структура данных какая была, такая и осталась.

Более того. Основной бинарный шматок памяти (который как в памяти сидит, так и сидит) вообще ни на бит не изменился. Просто деталями реализации удалось выпилить 3 раза. Каким образом. Чтобы чуть лучше понимать, каким образом, надо понимать, что делает критичный код.

Ты ему даешь на вход словоформу — он возвращает так называемый набор лемм (корневых форм). Внутри у него то, что авторы aot'a называют «конечный автомат», в принципе, похоже на обычное дерево (которое trie), как два брата близнеца. На каждом уровне ряд значений и указатель на следующий уровень.

Все это достаточно приятно запаковано. Многие сотни мегабайт данных пакуются в 10-20 мегабайт аккуратно упакованного суффиксного дерева.

HighLoad++

История любви

1.	ref,	wall 11.9		
2.	magic1,	wall 10.1,	1.178x relative,	1.178x total
3.	magic123,	wall 9.3,	1.086x relative,	1.279x total
4.	currpath,	wall 8.0,	1.162x relative,	1.487x total
5.	ndformsort,	wall 6.9,	1.159x relative,	1.724x total
6.	fastuc,	wall 6.3,	1.095x relative,	1.888x total
7.	dwordres,	wall 5.9,	1.067x relative,	2.016x total
8.	ptrres,	wall 4.2,	1.404x relative,	2.833x total
9.	manrecurse,	wall 4.0,	1.050x relative,	2.975x total

История любви была вот такая. Начинал я с 12-ти секунд в аот билд, закончил четырьмя секундами в аот лог. Чисто для понимания — вот эти 12 секунд и 4 секунды, ради чего все делалось, на фоне всей остальной индексации, которая занимала 5 секунд. 5 секунд на индексацию и 12 секунд на морфологический словарь меня как-то морально беспокоили. Поэтому я предположил, что можно быстрее, и, действительно, можно быстрее суммарно в 3 раза. Недлинным рядом из восьми шажков.

HighLoad++

Шаги 1, 2

- +17.8% (magic1)
 - Выкидываем 1-буквенные “слова”
 - Выкидываем слова “нерусские”
- +8.6% (magic23)
 - Выкидываем наиболее частые 2/3-буквенные
 - И ведь всего-то 18 слов

Шажок № 1. Тупой, как дрова, и всегда работает. Ищем в коде fastpath — тот путь кода, который проходит, грубо говоря, в 90% случаев (на самом деле, меньше в том конкретном контексте, но неважно) — и закорачиваем его. В случае аот’я и индексации в полнотекстовый индекс таким fastpath’ом, очевидно, является индексация суперкоротких слов — предлогов из 1-2 букв, всяческих коротеньких слов из 2-3 букв.

Строим список стоп-слов, забиваем проверку на однобуквенное, но нерусское слово и на 12 наиболее частых двухбуквенных, 6 наиболее частых трехбуквенных предлогов прямо в код. Причем забиваем их, как положено, не сравнением строк, а двумя чтениями двух байт и сравнением целочисленной константы с word’ом. Тот самый switch против if’я, между прочим. Тот самый switch быстрее, чем if.

Внезапно +18% — за счет того, что выкинули однобуквенные слова, за счет того, что закоротили один из наиболее быстрых путей. Внезапно еще 8,6% за то, что закоротили один из часто используемых путей, когда суперчастый предлог «на» вообще не надо обрабатывать. Он долго и нудно обрабатывается, а на выходе он тебе отдает словоформу «на» для предлога «на». Закоротив этот код получаем 9%. Мелочь, а приятно.

HighLoad++

Шаг 3

- Было


```
std::string currentPath;
DoRecursiveStuff ( currentPath, ... );
```
- Стало +16.2% (currpath)


```
BYTE sPath[128];
DoRecursiveStuff ( sPath, 0, ... );
```

Следующий шаг № 3. Где-то в дебрях есть такой пример кода, где в какую-то рекурсивную функцию передается по ссылке (не копированием) обычный, стандартный std::string. Замена, которая вместо std::string начала туда-сюда гонять ссылку на байтовый буфер и длину этого байтового буфера, каким-то чудесным образом дала +16%. Честно говоря, я даже заленился разбираться, каким именно. Просто выкинул и забыл.

Очевидно, в случае передачи `string`'а, `string` оказался недостаточно тонким. Байтовые `pointer`'ы и длина оказались еще тоньше, чем `string`.

h! HighLoad++

Шаг 4

- Стало +15.9% (`noformsort`)


```
vector<int> indexes; string currentBest;
Get ( indexes );
for ( int i=0; i<indexes.size(); i++ )
    CheckAndUpdate ( currentBest );
return currentBest;
```

Совершенно тот же трюк в другом месте. Был вот такой паттерн в коде. Набиваем вектор каких-то результатов чём-то, сортируем его — возвращаем самый лучший результат. Честно говоря, это уже был интеграционный код. Переписал в другую форму.

h! HighLoad++

Шаг 5

- Был фарш `RmIMakeUpper`, `FilterSrc`
- Приводило регистр, заменяло букву Ё
- Стала прегенерация `BYTE m_UC[256];`
`while (*p) { *p = m_UC[*p]; p++; }`
- +9.5% (`fastuc`)

Вместо того чтобы возвращать толстые значения, строки, возвращаем индексы. Потом внутри этого самого `CheckAndUpdate` генерим те же самые строки, которые раньше возвращали через `vector`, и каждую сравниваем с текущим лучшим значением, каждую возвращаем и так далее.

Казалось бы, ничего не изменилось. Но +16%. За счет того, что туда-сюда между функциями в программе гоняется меньше данных. Вместо вектора строк гоняется вектор `int`'ов. Каждая конкретная строка генерится в махоньком буфере. Вместо того чтобы разосраться в память сотней строчек по 16 байт, мы каждую новую строчку просто генерим на месте локально.

Там существенно менее чудовищные цифры. Тем не менее, очередные 16%. Тем же самым фокусом передаем туда-сюда меньше данных.

h! HighLoad++

Шаг 6

- Гонялся и возвращался `vector<WORD[3]>`, получаемый распаковкой некоего `DWORD`
- Заменил на `vector<DWORD>`, в нужных местах добавил распаковку на месте
- +9.5% (`dwordres`)

Шаг № 5. Совсем тупой, но неожиданно хороший. Там была очень глупая функция привода в русский регистр, которая была очень кастомизирована в зависимости от разных опций.

Дескать, если такая опция стоит, то вот так приводим, а если такая — то вот так. Она достаточно неплохо была написана для функции, которая кастомизирована. Но трансляция по один раз посчитанной байтовой таблице дала свои 10%. Тут тупо упростиł код. Убрал код, который был очень общий, заменил его примитивным, зато максимально быстрым фильтром по таблице.

Шаг 6

- Гонялся и возвращался `vector<WORD[3]>`, получаемый распаковкой некоего DWORD
- Заменил на `vector<DWORD>`, в нужных местах добавил распаковку на месте
- +9.5% (`dwordres`)

Шаг № 6. Опять тот же самый паттерн. Туда-сюда где-то внутри программы, туда-сюда гонялся вектор между функциями из (казалось бы) махоньких структур. 3 word'a, 6 байт. Заменить его тупо на `vector<DWORD>` — внезапно +10%. Хотя, казалось бы, он крохотный. 6-байтные структуры. Более того, их там не 10 миллионов, их там максимум 10-20 штук. Тем не менее, оказалось, что гонять по 4 байта на 10% выгоднее, чем по 6.

Шаг 7/8

- Результат писало в `vector<DWORD> & Infos`
- Заменил на `DWORD[12]`, с маркером конца
 - Максимальная длина результата 6
- +40.4% (`ptrres`) !!!
- `vector<DWORD> g_res + g_res.reserve()` толк тоже давали, но меньше (очевидно)

Почти последний шаг. Результат всех этих и не только телодвижений внутри рекурсивной процедуры обхода дерева, конечного автомата, trie — как угодно его назови — записывался в обычный советский STL вектор. Заменить его на статический DWORD на стеку, с маркером конца (там, где список кончается, то ли 0, то ли -1 — не помню) дало внезапно +40%.

На самом деле, ничего внезапного, потому что вектор динамический и постоянно туда-сюда — «распредели память, отдав память, распредели память, отдав память». «Алиса, открути гаечку, закрут гаечку». Известный анекдот.

Что интересно. Если остаться в рамках любимого STL, не заменять его на статический буфер, просто вынести его в глобальную переменную и сделать `reserve`, мгновенно обосрав всю многопоточность, которая иногда бывает нужна, толк, в принципе, есть, но он существенно меньше. 40% не удавалось добиться. 20-30%, по-моему, выжималось, но не более того.

Шаг 8/8

- Развернул inner loop из рекурсии в цикл
- Было
`int Count = GetChildrenCount(NodeNo); for ...`
- Стало +5% (manrecurse)
`int iChild[MAX_DEPTH], iChildMax[MAX_DEPTH];
while (iLvl>=0) while (iChild[iLvl]<iMax[iLvl]) ...`

Последний шаг. Честно говоря, самый большой, потому что все предыдущие изменения были, грубо говоря, односторонние. Каждый diff на каждом шаге менял по 20 строк. Этот, по-моему, поменял 50. Самый-самый inner loop, когда совсем ничего уже придумать не мог, вместо того чтобы оставить рекурсивные функции, я развернул в цикл.

Там для каждой Node на определенном уровне мы получали число детей и обходили всех детей. Этот самый обход — я выпил стек вручную, по большому счету. Вот этот iChild, iChildMax — это состояние функции в точке вызова. Выпил вручную, развернул рекурсию, свои 5% выкурил из кода.

Неявный шаг номер 0

- BENCH, BENCH, BENCH
- Как следствие profile, profile, profile
- Черновик, замер, чистовик
- Десяток чистовиков был закоммитан
- Десяток черновиков сразу выкинут

На этом мне надоело, да и презентацию пора заканчивать. При всем при этом постоянно присутствовал неявный шаг номер 0. «Benchmark'ять, benchmark'ять, benchmark'ять». Постоянно benchmark'ять, постоянно profile'ить. Попробовали, стало быстрее или нет, запрофайлили, кто там сегодня в top'e.

Цикл «набросали черновик, сделали замер, если замер показал, что да, эффект от оптимизации есть» был проделан от 30 до 100 раз. В итоге десяток чистовиков тут описанных было закоммитано. Десяток — потому что некоторые из шагов делались в 2-3 подшажка. Десяток черновиков, которые пробовали всякое, другое, разное, просто сразу выкинут. Цикл «быстрый черновик, быстрый замер» показывает, что толку нет.

Сводка фокусов

- Душим fastpath, прямо в коде
 - Шаги 1, 2
- Душим RAM/stack pressure аргументов
 - Шаги 3, 4, 6, 7, 8
- Душим сложность, тупое упрощение влоб!!!
 - Шаг 5

Какая сводка разнообразных фокусов (я догадываюсь, что все это в голове не укладывается, и чтобы потом по слайдам восстанавливать). В данном конкретном примере видов фокусов всего-навсего 3.

Придушили fastpath. Точнее, сделали fastpath. Придушили очень сложные расчеты на очень частых путях исполнения кода. Это те самые шаги 1 и 2.

Существенно придушили давление на память и на stack, которое, в основном, возникало из-за гонения туда-сюда всяких аргументов. Это подавляющая часть шагов, как выяснилось, в данном случае.

В одном месте тупо удушили очень сложный код, заменили его на простой табличный look up. В итоге, напомню, boost в 3 раза и общее время индексации, соответственно, в 2 раза.

h++ HighLoad++

Сводка других фокусов

- Душим лишнюю индирекцию
- Душим аллокации, пулим всякое
- Душим использование RAM (локальность!)
- Душим “плохие” общие структуры
- Душим arch-specific фокусы (switch, sse, memb-pressure, for-if, ptr-walks, movzx, ...)

В природе есть еще миллион разных фокусов, которые можно делать, когда ты занимаешься оптимизацией. Убирать лишнюю индирекцию при обращении к тем или иным структурам данных, пуливать аллокации, писать аллокации вручную, перекладывать данные так, чтобы они более локально лежали. Удушивать «плохие» общие структуры, типа хеша строки и вектора, заменяя их на совсем простые варианты, хорошо заточенные под конкретную задачу.

В конце концов (last, but not least) — заниматься всякими специфичными фокусами, которые специфичны для архитектуры. Те самые замены switch'ей на if'ы, вынос if'а за цикл и так далее, и тому подобное. В конце концов была одна оптимизация, где замена байта на int дала эффект вполне измеримый. Почему? Потому что когда ты работаешь с байтом, компилятор генерит инструкцию movzx постоянно, а это медленнее, чем работать с int'ами. Замена байта на int дала вполне измеримые свои 2%.

h++ HighLoad++

Сводка третьих фокусов

- И еще несколько десятков всяких удивительных мелочей
- Из которых лично я...

Есть еще несколько десятков мелочей и удивительных фокусов, из которых я большую часть не знаю, а оставшиеся просто забыл, да и времени нет. Господин Тутубалин собирался их рассказать, но времени нет.

h++ HighLoad++

Мораль про приемы

- Приемов много, общее правило одно
- Идеально вообще не работать!!!

Мораль одна. Вот куча приемов. Вот куча знаний про RAM, про CPU, которыми надо обладать. Общее правило, гигантская манта одна — если можно не работать, значит, нужно не работать. Если можно код не исполнять, надо его не исполнять. Работу не делать, данные туда-сюда не перекладывать, в память ничего не писать, функции не вызывать, белое не надевать и на танцы не ходить.



Лежать себе в гамаке и ничего не делать. Идеальный код, исполняется максимально быстро.

HighLoad++

Мораль про приемы

- Но уж если работать приходится, то по минимуму
- Как именно схалявить — вопрос не всегда простой
- Приходится проявляться изворотливость!

Если работать приходится, то избежать этого, к несчастью, нельзя. Работать надо по минимуму. Как именно схалявить — вопрос сложный, приходится всячески изворачиваться и так далее. Самая важная деталь — это все про код. Это не инструкция для всех присутствующих. Это все про код.

HighLoad++

Мораль про процесс

- Запрофайлил, забенчмаркал, попробовал
- Намылил, прополоскал, отжал, повторил
- Каждый лишний 1% не лишний
- Курочка по зернышку...

Процесс достаточно простой. Очень нудный и очень незатейливый: запрофайлил, забенчмаркал, повторил. Я беру каждый лишний 1%. 20 раз взяв по 1% здесь, по 10% там — получается в итоге ускорить код от 3 до 7-10 раз. Курочка по зернышку — весь двор в говне.



Собственно, все.

Ведущий: Вопросы, пожалуйста, можно в кулуарах задать. Мы, к сожалению, не успеваем.

Анатомия баннерной системы

Артем Вольфтруб



HighLoad++

Анатомия баннерной системы

Артем Вольфтруб

Артем Вольфтруб: Коллеги, добрый день! Меня зовут Артем Вольфтруб. Я руковожу разработкой в компании «Gramant» и буду сегодня вам рассказывать про баннерные системы.

Вообще, идея этого доклада возникла у меня по результатам некоторой работы, хотя мы не занимаемся баннерными системами все время. Мы занимаемся самыми разными вещами. Но почему-то так случилось, что очень часто приходят запросы на различные баннерные системы. Мы сделали достаточно много сложных систем, в каком-то количестве принимали участие, консультировали и так далее.

(*Вопрос к слушателям*). Кто среди присутствующих сталкивался с баннерными системами? (*В зале поднимают руки*). Да, идея доклада была неслучайной. Наверное, это будет интересно.

Но случилась некая неприятность. Если раньше, на прошлых конференциях я просил мало времени, а мне давали очень много, то здесь получилось все наоборот. Я думал, что наконец-то тема, которая обширная, в ней можно много про что рассказывать, но время безжалостно урезали. Поэтому мне пришлось тоже урезать свой полет фантазии. Возможно, что-то я не успею осветить. Тогда будем общаться в кулуарах.



imho VI

- 50 млн. показов в день
- Система на базе Dart Enterprise
- Расширенная статистика

ValueCommerce

- 700 млн. показов в день
- Собственное решение
- Модель ориентированная на продажи

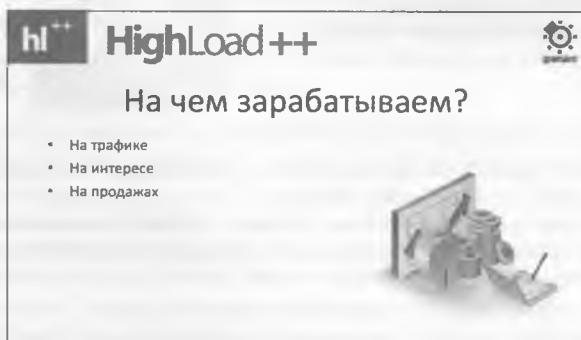
Для начала слайд пиара, чтобы говорить не о каких-то абстрактных системах, а о более или менее известных. Мы делали такие системы (ValueCommerce, Imho VI). Сделали такую систему (hh.ru). Она сейчас практически запущена.



Самопиар закончен. Дальше будет доклад. Этот слайд, на самом деле, иллюстрирует очень простую баннерку. Такая примитивная баннерка состоит как раз из трех вещей: баннер сервер, консоль управления («админка»), база данных и статистика.



На самом деле, все намного сложнее. На этом слайде — чуть-чуть более развернуто, хотя тоже не до конца. Отдельные компоненты еще можно разворачивать и представлять более сложную структуру. Это уже ближе к реальности. Сегодня будем говорить, в первую очередь, про баннерные сервера, про статистику, про мониторинг. Про управление, про «админку» говорить не будем, потому что там, по-моему, все достаточно просто, во-первых. Во-вторых, она очень индивидуальная. В зависимости от системы пересечений не очень много.



Сначала давайте определимся с тем, какие баннерки бывают. Вернее, не какие они бывают, а исходя из чего выбирается та или иная архитектура. По нашей практике есть 3 модели бизнеса, на которых делаются баннерки.

Первая модель — это когда баннерки зарабатывают на трафике. Есть какой-то большой ресурс, или ресурсы, или какое-то количество сайтов с большим трафиком, и на них тупо размещается реклама. Это как медийная реклама. Там практически неважно, кому она продается. Важно, чтобы она крутилась. Какие-то промо-кампании, брендингование и так далее.

Следующий самый распространенный тип. Когда продается интерес пользователя — показы либо клики.

Третий случай, который, насколько я знаю, в России практически не представлен. Когда продается продажа. Баннерная система зарабатывает на конечных продажах.

The screenshot shows the HighLoad++ landing page. At the top left is the logo 'hl++'. In the center is the title 'HighLoad++' in large, bold, sans-serif font. To the right is a small circular icon with a gear-like pattern. Below the title is the subtitle 'Ориентируемся на трафик' (Oriented towards traffic). Underneath this, there is a bulleted list of features:

- Нагрузка на баннерные сервера
- Простая логика показов
- Простая статистика

On the right side of the page is a small image of a computer monitor displaying a dark interface with the word 'Sales' visible. The overall design is clean and professional.

Соответственно, в зависимости от того, какая модель бизнеса используется, у баннерки есть та или иная особенность. Скажем, если мы делаем баннерку, которая ориентируется на трафик, то она, конечно, характеризуется большой нагрузкой, как и любая другая баннерка. Но тут важно, что логика показов очень простая. Это практически просто отдача статического контента без каких-то сложных правил таргетирования, без лимитов, без каких-то учетов уникальных пользователей, потому что это никого особо не интересует. Важно показать рекламу. Кому она была показана — уже не так важно.

The screenshot shows the HighLoad++ landing page. At the top left is the logo 'hl++'. In the center is the title 'HighLoad++' in large, bold, sans-serif font. To the right is a small circular icon with a gear-like pattern. Below the title is the subtitle 'Ориентируемся на интерес' (Oriented towards interest). Underneath this, there is a bulleted list of features:

- Сложный таргетинг
- Борьба с накрутками
- Сложная статистика
- Уникальные пользователи

On the right side of the page is a small image showing a person's hand interacting with a computer keyboard. The overall design is clean and professional.

Следующий тип. Ориентируемся на интерес. Здесь все гораздо сложнее. Это самый распространенный тип, естественно (особенно в России). Здесь, как правило, есть таргетинг. Обязательно возникают вопросы борьбы с накрутками. Обязательно сложная статистика. Как правило, присутствуют уникальные пользователи, что доставляет немало проблем.

hl++ HighLoad++

Ориентируемся на продажи

- Мониторинг продаж
- Интеграция с рекламодателями
- Сложный биллинг

Последнее — баннерки, которые ориентированы на продажи. Тут самый сложный компонент — это мониторинг продаж. Сложность здесь в том, что нужно интегрироваться с рекламодателями. Не всегда удается внедрить им какой-то свой код. Иногда приходится у них забирать данные о продажах, каким-то образом сопоставлять с теми событиями, которые происходили в баннерке.

Там, как правило, очень сложный биллинг. Во-первых, нужно считать все комиссии по тем транзакциям, которые были. Кроме того, как правило, рекламодатели используют достаточно навороченную модель комиссий. Делятся по категориям товаров, по объемам, по пользователям, по регионам. Очень много правил, которые надо учитывать.

hl++ HighLoad++

Баннерный сервер

- Mission critical компонент в любой баннерке
- Наиболее нагруженный компонент

- Простота
- Слабая связанность
- Независимость
- Масштабируемость
- Принцип «не молчать»

Начнем с баннерного сервера. Понятно, что в любой баннерке это то, что называется *Mission critical* компонент (основной компонент, без которого баннерка работать не будет, это одновременно и наиболее нагруженный компонент).

Исходя из этого какие требования предъявляются к баннерному серверу.

- Простота. Чем проще он работает, тем лучше. Простота — значит, скорость, в первую очередь.
- Слабая связанность. Нужно стараться, чтобы баннерный сервер как можно меньше обращался к другим компонентам нашей баннерки. В идеале — никуда не обращался, но это не всегда получается. Чем меньше связанный, тем лучше.
- Независимость от других баннерных серверов. Это тоже очень важно, чтобы каждый баннерный сервер был абсолютно независим от других, поскольку это сильно влияет на масштабируемость.
- Масштабируемость — основное требование для баннерных серверов. Если бизнес идет хорошо, то нагрузка растет быстро, и масштабироваться надо тоже быстро.
- Принцип «не молчать». Смысл его в том, чтобы баннерный сервер в любом случае, в независимости от того, что происходит, отдавал какой-то результат за конечное вре-

мя. Это нужно, чтобы пользователи или системы, которые ожидают ответа баннерного сервера, не показывали какое-то сообщение об ошибке или — того хуже — тайм-аут и так далее. Важно сделать так, чтобы ваша баннерка давала ответ за какое-то конечное время. Даже если она не может показать нужный баннер. Пусть лучше отдаст какой-то баннер-заглушку, но, по крайней мере, это будет какой-то ответ.



Немного поговорим о стратегиях обновления баннерных серверов. Есть две. Одна стратегия – так называемая Pull. Баннерные сервера запрашивают данные либо из базы данных, либо из какого-то промежуточного компонента (API и так далее), запрашивают данные рекламных компаний, которые надо показать.

На мой взгляд, это неправильная стратегия. Тем не менее, она присутствует. Я сталкивался несколько раз, когда люди ее использовали. Аргументы обычно были такие: это прямая связь между баннером и базой данных или системой хранения, системой управления компаниями. Нет никаких лишних звеньев в цепи. Очень просто добавить новые сервера. Они говорили: «Мы ставим новый сервер, выкатываем на него код баннерки, запускаем его — и все, он при старте считывает все данные, начинает показывать». В принципе, действительно, это так.

Еще был аргумент: «Можно иметь разное содержимое на разных баннерных серверах». Скажем, одни баннерные сервера отдают картинки, другие — флеш-ролики, третий — видео. Не знаю, зачем это нужно. Но иногда встречается такое требование.

Тем не менее, такая модель очень сильно усложняет код баннерного сервера, потому что весь этот механизм синхронизации должен располагаться на баннерном сервере. Она не позволяет обновить все сервера сразу. Но если у нас случился какой-то креш, мы переехали в новый дата-центр или еще что-то сделали, нам приходится обновлять каждый баннерный сервер в отдельности.

Что самое главное — она не позволяет нам понять, что сервер упал. Конечно, можно поставить систему мониторинга на каждый баннерный сервер. Но, в общем-то, если баннерный сервер упал, то система мониторинга, скорее всего, тоже с ним упадет. Мы не узнаем о том, что какой-то сервер упал.

Увеличивается неконсистентность данных, поскольку каждый сервер запрашивает обновления у хранилища базы данных в произвольный период времени, тем самым может оказаться, что в один и тот же момент времени информация на них разная. Это тоже не есть хорошо, если у нас используется принцип однородности баннерных серверов.

hl++ HighLoad++

Стратегии обновления. Push

- Единая точка мониторинга
- Возможность обновить все сразу
- Обновления по расписанию
- Нужно регистрировать новый сервер
- Создается точка отказа
- Сложно поддерживать разный контент на баннерных серверах

Вторая стратегия — это Push-обновления, когда мы через какой-то специальный модуль синхронизации обновляем сразу весь пул баннерных серверов. На мой взгляд, этот подход более правильный, более хороший, потому что у нас есть единая точка мониторинга. Если в процессе обновления какой-то баннерный сервер не отвечает, мы это сразу знаем. Мы можем сразу обновить все сервера и можем задать расписание обновления. Это, на самом деле, очень важно, если мы знаем о каких-то событиях, которые происходят в нашей «админке».

Чуть позже я буду говорить о лимитах. Лимиты могут высчитываться на основе статистики. Если мы знаем тот момент, когда статистика готова, в этот момент хорошо бы обновить информацию на баннерных серверах. Если мы делаем обратную операцию (Pull), то мы не знаем, готова статистика или нет. Может быть, она будет готова через секунду после того, как мы спросим апдейт. Тем самым мы упустим возможность получить актуальную информацию.

Тем не менее, есть некоторые минусы (на мой взгляд, несущественные). Первый — нужно зарегистрировать новый сервер в этом модуле синхронизации. Он должен знать обо всех баннерных серверах, которые стоят в пуле. Нужно, но, по-моему, не очень смертельно.

Единая точка отказа, что тоже часто приводят в качестве минуса. Да, хотя, на самом деле, ее легко мониторить. Можно сделать несколько модулей синхронизации. Можно поднимать его автоматически. При падении это, на мой взгляд, тоже спорный аргумент.

Насчет поддержки контента на разных баннерных серверах. Действительно, если у нас такая задача стоит, то придется поддерживать это на уровне модуля синхронизации. Но мое личное мнение — этого не нужно делать. Я ни разу не встречал ситуацию, когда действительно нужно было бы иметь разный контент на разных баннерных серверах. Лучше, чтобы они были независимы. Это сильно упрощает и масштабирование, и поддержку, и избавляет от ряда других проблем.

hl++ HighLoad++

Синхронизация

Инкрементальная <ul style="list-style-type: none"> ▪ Экономия трафика ▪ Уменьшение задержки обновления 	Полная <ul style="list-style-type: none"> ▪ Одномоментное переключение версии данных ▪ Простота реализации ▪ Необходима в любом случае
---	--

Раз уж мы начали говорить о синхронизации, чуть-чуть затронем, какая она вообще может быть. Бывает двух видов — инкрементальная и полная.

Инкрементальная означает, что мы обновляем не всю информацию, а только ту, которая изменилась за время с момента последнего нашего обновления. Это, с одной стороны, экономит нам трафик и уменьшает задержки обновления. Собственно процесс синхронизации занимает меньше времени, нежели в случае полной синхронизации.

С другой стороны, такой подход чуть более сложен, потому что нам приходится следить за версионностью объектов на баннерных серверах. Мы не можем одномоментно переключить версию.

В случае полной синхронизации мы можем создать в памяти баннерного сервера копию текущего состояния нашей базы данных и в один момент переключить то, что он до этого показывал, на то, что мы ему обновили. В случае инкрементальной это сделать сложнее.

Понятно, что полную синхронизацию проще реализовать. Она в любом случае понадобится. Например, установили новый сервер в пул, и надо налить на него все данные. Или, например, сервер упал и пролежал 2 дня, потом у него заменили диск (или не диск — память добавили), и нужно его обновить. В случае инкрементальной могут возникать ошибки, потому что очень сложно восстановить всю цепочку версий. Гораздо проще с нуля все налить.



Немного скажу про хранение медиа данных. Здесь, в принципе, ничего такого сложного нет. Знаю 3 варианта.

1. Когда мы храним в файловой системе баннерных серверов — достаточно простой вариант, ни от чего не зависимый. В момент синхронизации мы каким-то образом копируем на баннерные сервера собственно сами баннеры и раскладываем их там каким-то образом. Скорее всего, все будут использовать какие-то стандартные средства Unix'a. Про остальные можно даже не говорить.

2. Отдельное хранилище — это вариант, когда мы выделяем какой-то специальный сервер, общаемся с ним, например, по DAV-протоколу. Выкладываем туда баннеры. Конечные пользователи загружают к себе эти баннеры по протоколу HTTP. Тут тоже все достаточно просто. Единственное, что нужно, это поддержать этот протокол обмена по WebDAV, хотя есть куча готовых клиентов, и большой сложности это не представляет.

3. Распределенное хранилище. Достаточно популярный в последнее время вариант. Особенность если у нас речь идет о видео, о больших картинках или, например, если нам нужно покрыть какой-то большой регион в России. Гораздо выгоднее использовать CDN с точки зрения цены трафика, чтобы пользователю, который живет во Владивостоке быстрее доставлять трафик, который лежит где-то рядом (хотя бы в Новосибирске, а не в Москве). Это хороший вариант, который, безусловно, надо рассматривать, если у вас стоит задача охвата большой аудитории либо возникает проблема стоимости трафика, если его много.

hl ++ HighLoad++

Лимиты

- Одна из самых важных для бизнеса фич
- Бережливые менеджеры и расточительные разработчики

Типы лимитов

- По рекламной кампании
- По пользователю

Переходим к лимитам. Если мы говорим о баннерке, которая продает интерес пользователей, там всегда возникает вопрос лимитов. Продаются обычно либо показы, либо клики. Чаще у нас в России продаются, по-моему, показы. Хотя сейчас потихоньку все движется в сторону кликов, что, на мой взгляд, правильно.

Для бизнеса это одна из самых важных фич. Рекламодатели очень охотно ее покупают, потому что они понимают, за что они платят. Они платят не за какой-то абстрактный показ рекламы. Они платят за конкретных пользователей, которые перешли к ним на сайт.

Тут всегда возникает вопрос. Есть менеджеры, которые продают эту рекламу, которые говорят: «Нам нужно очень внимательно следить за лимитами. Мы не можем перекручивать рекламные кампании. Мы должны завершать все кампании, как только у нас лимит показов или кликов исчерпан». Есть разработчики, которые начинают предлагать разные компромиссы: «Может быть, полчаса задержка. Много ли мы там за полчаса перекрутим. Если полчаса много, то какую можно задержку ввести?». Начинается спор и торговля.

При этом есть два типа лимитов.

- Один тип — по рекламной кампании, когда есть бюджет. Например, количество кликов в кампании, за которую заплатил рекламодатель.
- Второй тип лимитов, который тоже очень любят, лимит показов по какому-то конкретному пользователю. Лимит кликов сделать вряд ли удастся. Показов. Показать баннер одному пользователю не больше, чем x раз.

hl ++ HighLoad++

Обновление статистики

- Упрощает работу баннерного сервера
- Уменьшает время ответа
- Нельзя сделать без задержки, будут перекрутки.
- Требуется очень быстрая обработка статистики
- Сложно считать показы уникальным пользователям

На самом деле, это разные лимиты, которые, как выясняется, нужно по-разному реализовывать. Есть два решения, как можно реализовать лимиты.

Первое — обновлять через статистику. Мы откручиваем рекламу, собираем статистику. Смотрим, сколько мы открутили, и эти обновленные лимиты передаем на баннер-сервер в процессе синхронизации. Это упрощает работу самого баннерного сервера, потому что ему ни откуда не нужно доставать эти лимиты, они приезжают вместе с нашими данными во время очередной синхронизации.

Это уменьшает время ответа, потому что все данные находятся у banner engine'a. Ему не нужно обращаться никуда вовне, чтобы получить эти данные. Но это нельзя сделать без

задержки, потому что всегда будут перекрутки. Обновлять статистику в реальном времени мы не можем.

Также это требует очень быстрой обработки статистики. Если у нас много данных, то задержка может быть существенной (может быть 5 минут, а может быть и полчаса). Когда мы сделали систему, где было несколько сотен миллионов показов ежедневно, задержка статистики была порядка получаса. Там, к счастью, лимитов не было. Но, в принципе, это было большой проблемой, потому что за полчаса можно показать очень много.

Такое решение не позволяет считать показы по уникальным пользователям, потому что этих данных настолько много, что в реальном времени их считать не удается. Можно, конечно, делать какие-то прогнозы (посчитать какую-то порцию и экстраполировать это на число пользователей). Но если мы хотим точности, это не посчитать в реальном времени.

The screenshot shows a section titled 'Общий кеш' (General Cache) with the following bullet points:

- Актуальные данные
- Отсутствие перекруток
- Дополнительная точка отказа
- Нужно синхронизировать со статистикой
- Данные могут вымываться

Второе решение — использовать общий кеш. Как вариант — `memcached` или другие решения. Это позволяет нам иметь актуальные данные. В момент показа мы сразу инкрементируем счетчики — и баннерные сервера с ними работают. Это тем самым гарантирует нам отсутствие перекруток. Но это дает нам дополнительную точку отказа, потому что `memcached` (и вообще кеш) может сломаться, упасть.

Еще одна проблема — нужно синхронизировать эти данные со статистикой, в какой-то момент передавать информацию (она может расходиться) либо полагаться на те логи, которые мы обрабатываем. Но в идеале нам нужно актуальные данные из `memcached` передавать в нашу статистику, чтобы потом эту информацию обратно вернуть баннеру.

Еще одна проблема. Если, в частности, мы используем `memcached`, то данные могут вымываться. Если их станет слишком много, то какие-то данные будут теряться. В этом случае мы вообще не найдем нашего значения.

The screenshot shows a section titled 'Лимиты' (Limits) with the following bullet points:

- Не используйте без нужды
- По возможности используйте статистику
- Готовьтесь к падению кеша
- Разные стратегии для разных типов лимита

Общий совет и рекомендация по лимитам такова, что если вам не нужно их использовать, лучше их вообще не использовать. Так редко бывает, но вдруг.

По возможности, лучше использовать статистику (по крайней мере, когда мы говорим о лимитах по кампаниям). Все-таки лучше пожертвовать пятиминутной задержкой, если у вас не полчаса, и тем самым упростить работу баннерного сервера.

Нужно ожидать падения кеша. Нужно эту ситуацию обрабатывать и не допускать ситуации, когда данных в кеше не будет, и баннерный сервер не будет знать, что с этим делать.

Лучше использовать две стратегии для разных типов лимитов. Скажем, наш последний подход состоит в том, чтобы лимиты по компаниям обновлять через статистику, а лимиты по показам для конечного пользователя хранить в кеше и оттуда доставать (в статистике вообще не нужны лимиты по пользователям, только для показов).

hi ++ HighLoad ++

Таргетинг

- Занимает большую часть времени обработки запроса
- Получение значений не должно быть частью баннерки

Таргетинг. На самом деле, когда я сделал этот слайд, я понял, что рассказывать про него особо нечего. Все достаточно просто. Есть только два момента, которые нужно учитывать.

Первый момент состоит в том, что по нашим расчетам выбор таргетинга, проверка всех условий — это большая часть времени обработки запроса на стороне баннерного сервера. Поэтому лучше использовать, конечно, не простой полный перебор и цикл, а какой-то более разумный алгоритм. В частности, мы используем побитовые маски, чтобы быстро выбирать по имеющимся условиям нужный баннер. Это не единственный алгоритм, просто как пример.

Второй момент. Конечно, стоит стараться ограничить число параметров таргетирования. Чем больше параметров, тем сложнее их будет считать. Очень важно договориться о том, где будут храниться параметры.

Ни в коем случае не надо допускать ситуации, когда вам говорят: «Пришел запрос от пользователя. Там есть IP-адрес. По IP-адресу определите регион и сделайте таргетинг по региону». Это очень плохая ситуация. Баннерный сервер должен получать все значения таргетинга в момент поступления запроса на показ баннера, иначе драгоценное время будет потрачено впустую. Лучше пусть вы сами либо кто-то еще, кто передает вам данные, сделают эту работу. Но тем самым вы сэкономите драгоценное время вашего баннерного сервера.

hi ++ HighLoad ++

Учет событий

- Хранение в файлах, а не в памяти
- Никакой предварительно обработки на баннерном сервере
- Вообще никакой обработки

Чуть-чуть про регистрацию событий. Тут тоже все достаточно просто. Но на практике была ситуация, когда люди делали большую ошибку и потом с этим мучались. Ошибка заключалась в том, что, во-первых, они логи, которые накапливаются на баннерном сервере, держали в памяти. Во-вторых, из баннерного сервера обновляли напрямую в базе данных. Это очень плохой вариант.

Первый минус — если падает баннерный сервер, то все ваши логи теряются. Второй минус — если падает канал с базой данных, то вообще непонятно что происходит. Третий минус — обращение к базе данных по сети — это дорогостоящая операция, которую лучше не использовать.

Самый простой паттерн — вначале храним в памяти. Как только память буфера накапливается, скидываем файл. Как только файл превышает определенное значение, делаем ротацию и те файлы, которые уже готовы, мы потом заберем нашим модулем, который считает статистику.

Конечно, никакой предварительной обработки не нужно делать на баннерных серверах. Пусть это делает отдельный модуль, который будет знать, как лучше обработать статистику. Он собирает ее с разных баннерных серверов, и вы просто опять же усложните логику баннерного сервера. Этого не надо делать.



Несколько вариаций на тему, что еще может быть. Тоже из практики — какие возникают запросы от менеджеров, которые продают рекламу.

Равномерное распределение по времени суток (хотя бы показов). И то это большая проблема, потому что если у нас статические места, где без таргетинга, без всего, там все достаточно просто. Мы можем это посчитать.

Если у нас таргетинг, то понятно, что разные запросы выпадают с разной частотой. По-хорошему, это надо делать на основе статистики. Это сильно увеличивает сложность этого хранения. Если нам надо сохранить статистику по различным параметрам таргетинга (причем не просто по отдельным параметрам, а еще и по их комбинациям), то это в геометрической прогрессии увеличивает объем данных. Этого надо опасаться. Если уж так случилось, то нужно стараться количество параметров таргетинга, которые будут учитываться, хотя бы здесь сократить. Чем их будет меньше (не количества самих параметров, а их возможных значений), тем меньше данных будет у вас в статистике.

Динамический выбор лучшего баннера. Здесь, в принципе, не очень сложно это сделать. Но смысл в чем. Есть несколько баннеров, которые загружает рекламодатель. Спустя какое-то время, когда у нас накапливается статистика, мы выбираем тот, у которого выше CTR, тем самым повышаем вероятность клика на него и делаем хорошо рекламодателю. В принципе, проблемы особой нет, но опять же надо смотреть: как бы так не получилось, что, скажем, в случае баннеров таргетингом это может привести к тому, что мы не сможем выбрать нужный баннер.

Принудительная открутка. Когда у нас есть лимиты на показы (например, 10 тысяч показов, и нам нужно выбрать их за сутки). Речь идет о том, что динамически подкручивать веса на-

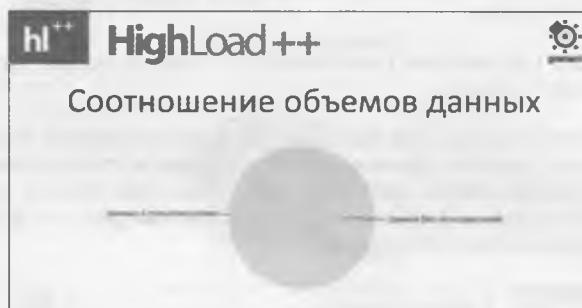
ших баннеров и показывать тот баннер, который хуже всего откручивается. Здесь проблема такая, что модуль, который это делает, может вступить в конфликт с самим собой. Для двух баннеров, которые крутятся на одном и том же месте, будет постепенно увеличивать веса, и они будут друг с другом бороться. За этим надо следить.



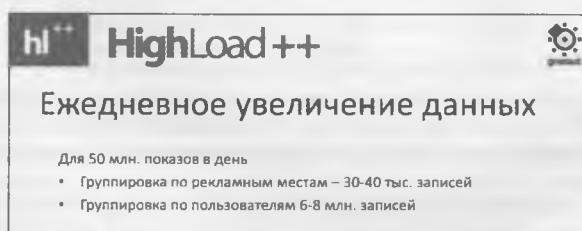
Про статистику. Есть два основных типа данных.

- Самая простая статистика, где показы, клики, CTR'ы.
- Статистика по уникальным пользователям. Мы сохраняем информацию о каждом пользователе, сохраняем у него в куках идентификатор, и потом этот идентификатор отслеживаем.

Абсолютное большинство баннеров, которые делаются у нас, хотят статистику по уникальным пользователям, потому что она позволяет делать всякие бизнес-вещи, о которых я чуть дальше расскажу.



Соотношение объемов данных примерно такое, как отображено на слайде. В 2 тысячи раз данных с пользователями больше, чем данных без пользователей. Почему — потому что данные без пользователя сворачиваются легко по рекламному месту. Если у нас есть какое-то место на сайте, мы можем все показы с этого места тупо инкрементировать в одной строчке базы данных. Если у нас есть уникальный пользователь, делать мы этого не можем.



Для 50-ти миллионов показов в день ежедневное увеличение примерно такое:

- 30-40 тысяч записей, если мы группируем по рекламным местам без пользователей.
- 6-8 миллионов, если мы делаем это с пользователями.

В результате наша работа с такой системой приводила к тому, что ежедневно приходилось копировать, делать архив, партицию базы, которая хранила данные по уникальным пользователям, и складывать их в архив. Иначе она просто не работала.

hl ++ HighLoad ++

Что получаем взамен?

- Частотная характеристика
- Анализ пересечения аудитории
- Анализ параметров таргетинга
- Post-click анализ

Почему, собственно, все так хотят эту статистику? Потому что она дает массу приятных отчетов, которые действительно хорошие и интересные. В частности, можно посмотреть частотную характеристику.

Например, как росла аудитория рекламной кампании с момента ее старта. Может быть, в какой-то момент имеет смысл останавливать эту кампанию.

Можно смотреть пересечение аудитории. Пересечение одного и того же пользователя на разных сайтах, на разных баннерах.

Можно смотреть анализ параметров таргетинга. Это тоже здорово. Как я уже сказал, это нужно, чтобы планировать показы.

Самое важное — post-click анализ. Это все хотят, но мало кто делает. Когда мы отслеживаем поведение пользователя на сайте рекламодателя. Ставятся специальные маркеры, связанные с баннеркой. Дальше мы можем смотреть такие вещи, как глубина проникновения, количество открытых страниц, какие действия пользователь совершал — в общем, массу всяких вещей, которые маркетологи просто обожают.

hl ++ HighLoad ++

Обработка статистики

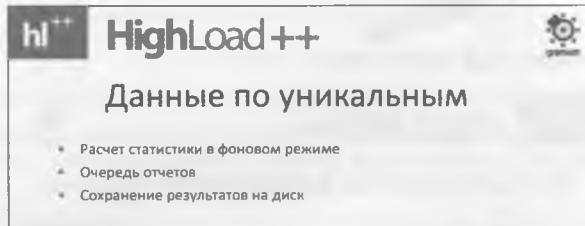
- Отдельный компонент системы
- Узел суммирования и фильтрации данных
- Быстрая загрузка
- Разделение баз данных
- Денормализация данных

По поводу обработки статистики. Хорошо бы это все выносить в отдельный компонент системы, делать специальный узел, который будет суммировать и фильтровать данные (то есть отсекать паразитный трафик, который случается, накликивание и так далее).

Очень важно быстро загружать эти данные в базу данных, не SQL-запросом, а желательно средствами базы данных типа SQL*Loader для Oracle или Copy для Postgres (какие-то стандартные утилиты).

Важно разделять базы данных. Отделять статистику, скажем, от данных рекламных кампаний и, тем более, от уникальных. Уникальные — это нежизнеспособная архитектура, если эти данные живут вместе с чем-то еще.

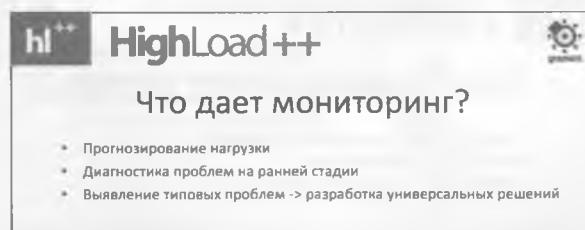
Для отчетов приходится денормализовать данные. Готовить для каждого отчета свою таблицу с данными, куда в процессе обработки статистики складывать ровно то, что нужно для построения отчета.



Для генерации отчета по уникальным даже то, что я привел на предыдущем слайде, не работает. Они не считаются в реальном времени, и приходится делать это в оффлайне. По нашей практике, самый простой вариант — делать все в фоновом режиме. Делать очередь отчетов, чтобы они запускались, а результаты складывались куда-нибудь на диск, чтобы два раза не пересчитывать один и тот же отчет. Можно в конце дня пересчитывать отчет по уникальным за день, можно делать это по запросу пользователя. Но важно, чтобы это делалось один раз.



Мониторинг. Это обширная тема, поэтому тоже очень быстро.



Зачем нужен мониторинг, никому, я думаю, рассказывать не надо. Все прекрасно понимают, что это и прогнозирование нагрузки, и диагностика, и — главное — выявление типовых проблем, которые позволяют создавать типовые решения для поддержки.

hl HighLoad++



Виды мониторинга

- Физический уровень
- Уровень приложения
- Бизнес уровень

По нашей практике 3 вида мониторинга имеют смысл.

hl++ HighLoad++



Мониторинг на физическом уровне

- Сеть
- Доступность сервера
- CPU
- Память
- IO
- Свободное место на дисках

Физический уровень — когда мы мониторим такие вещи, как сеть, доступность сервера, CPU, память, свободное место на диске.

hl++ HighLoad++



Мониторинг на уровне приложения

- HTTP Errors
- Response Time
- Актуальность баннеров
- Актуальность статистики
- Размер очередей

Мониторинг на уровне приложений — тоже стандартные, в принципе, вещи. HTTP-ошибки, актуальность баннеров, статистика, размер очередей и так далее.

hl++ HighLoad++



Мониторинг на бизнес уровне

- Число зарегистрированных событий за период
- Динамика рекламных кампаний
- Количество отфильтрованных данных

Самое последнее — мониторинг на бизнес-уровне, который зависит исключительно от логики приложений. Например, это может быть динамика кампаний, количество отфильтрованных данных. Если мы видим, что у нас за какой-то период резко падает число показов, значит, что-то происходит, надо на это тоже как-то реагировать. Это все тоже можно смотреть. Но опять же, это определяется уже конкретной системой.

hl++ HighLoad++

Чем измеряем

- Критериальная система Nagios
- Тренды Cacti
- Журналы Tenshi



Для измерения тоже все стандартно. В основном, это Nagios, для которого пишутся какие-то критерии, тренды Cacti, журналы Tenshi, графики, анализ логов. Это все делается в едином комплексе.

hl++ HighLoad++

Система мониторинга



The diagram illustrates a monitoring process. It starts with a stack of logs labeled "Logs and application servers". An arrow points to a central box labeled "Monitoring system". From there, arrows point to two figures labeled "Administrator" and "Analyst".

Выглядит это примерно так. Логи и информация о приложении и application серверах сливаются в систему мониторинга, которая на основе критериев генерирует какие-то события. Эти события обрабатываются командой системных операторов.

Причем для большинства событий уже заранее готовится специальная так называемая «книга рецептов», которая говорит, какую кнопочку надо нажать, если загорелась лампочка определенного цвета. Это позволяет сильно сократить нагрузку на системных администраторов и предоставить им возможность заниматься только действительно проблемами, с которыми справиться могут специально обученные инженеры.

hl++ HighLoad++

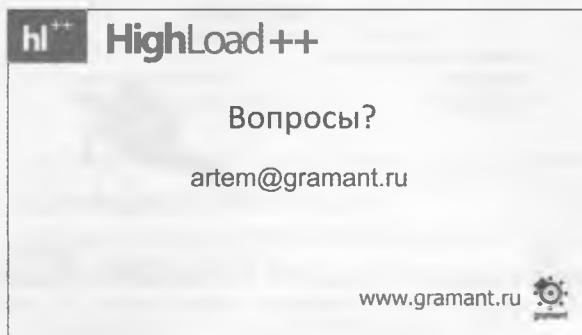
Поддержка системы

- Налаженная система мониторинга
- Наличие готовых лекарств
- Простота архитектуры
- Отлаженная процедура обновлений



Совсем чуть-чуть про поддержку. Главное, что дает хорошую поддержку, это наличие отлаженной системы мониторинга, наличие тех самых готовых лекарств, которые позволяют решать большинство проблем, не привлекая специалистов. Чем проще архитектура, тем проще поддерживать. Очень важно не забывать про отлаженную систему процедуры обновлений,

которая позволяет избежать множества проблем, связанных именно с выкаткой новых версий и с ошибками, которые могут быть заметны конечным рекламодателям, клиентам.



У нас, наверное, осталось минуты 3. Я готов ответить на вопросы.

Вопросы и Ответы

Вопрос из зала: Здравствуйте! Спасибо за доклад. Вопрос. На чем вы писали баннерные сервера. Как много нагрузки держит один сервер в секунду (запросов). Про статистику. База данных у вас — это какой-нибудь Hadoop или самописные вещи?

Артем Вольфтруб: Поскольку здесь упоминалось несколько систем, я возьму какую-нибудь одну для примера. В принципе, мы пишем на Java. Баннерные сервера тоже пишем на Java.

По нагрузке, если не ошибаюсь, это порядка полутора тысяч запросов в секунду на сервер без проблем.

По поводу базы данных — обычная реляционная база данных. Hadoop мы не используем. Я общался с докладчиком, который выступал в соседнем зале. Там они используют для статистики как раз NoSQL базу данных. Но, в принципе, это не дает им большого бенефита, потому что они также считают в оффлайне все запросы по уникальным. Это не было для нас узким местом (реляционная база данных).

Вопрос из зала: У меня вопрос следующий. Я говорю как человек со стороны, потому что баннерами никогда не занимался. Но в результате прослушивания доклада возник настойчивый вопрос по аналогиям. Вчера был доклад по системам сбора данных, сегодня — ваш доклад. Бросается в глаза очень большая аналогия задач. То же самое, что вы сейчас говорили по лимитам, в тех системах это уставки на телепараметры и так далее. Задачи настолько аналогичны. По показу дальше были эти аналогии.

Вопрос. Разработчики баннерных систем используются ли тот опыт, который наработан в параллельной области? У промышленной автоматизации опыт на пару-тройку десятилетий больше. Там существуют и архитектурные, и прочие решения.

Артем Вольфтруб: Вопрос, конечно, очень хороший. Я, к сожалению, вчера не смог присутствовать на докладе, о котором вы говорите. Но что касается наших задач, сделать эту систему — это не rocket science. Вопрос не в том, чтобы писать код. Это не очень долго. Вопрос в том, что там просто много данных. Скорее, вопрос железа и скорости обновления.

Здесь, мне кажется, что коллеги ничего принципиально лучше не сделали. Насколько я понимаю, что у них тоже какая-то своя система. Я, если честно, не знаю какого-то Open Source решения, которое бы позволяло решить проблему лимитов. Может быть, я просто не в курсе. Но опять же по нашей практике написать это не очень сложно.

Вопрос из зала: У меня простой вопрос. У вас есть какие-то системы античита или прогнозирования? Что-нибудь в этом духе.

Артем Вольфтруб: Хороший вопрос. Я его выкинул из доклада, потому что времени не было. На самом деле, тут есть два аспекта. Во-первых, борьба с накликыванием. Насколько я знаю, все, что используют (и мы, и наши коллеги), это стандартный набор критериев. Количество кликов с одного пользовательского IP-адреса в единицу времени, проверяет браузер, проверяется подсеть. Какой-то набор правил, который позволяет отсечь такой паразитный трафик. Это, конечно, никакая не стопроцентная гарантия. Но это основное. Я не знаю, может быть, что-то еще появилось.

Есть второй момент. Он связан с проверкой баннерки. Очень часто рекламодатели ставят такую задачу, чтобы каким-то образом проверить, насколько правильно баннерка показывает рекламу. Они купили, условно говоря, миллион показов и хотят убедиться, что действительно баннерка открутила миллион показов. Это делается с помощью вставки в баннерную систему (даже не в саму систему, а на сайт, где стоит баннер) кода с какой-то сторонней авторитетной баннерной системы (Double Click, или AdRiver, или какие-то еще системы). Такие задачи возникают постоянно. Это, в принципе, решается вставкой кода, который позволяет сравнить количество показов по версии AdRiver'a, например, и по версии нашей баннерки.

Вопрос из зала: (Неразборчиво, без микрофона).

Артем Вольфтруб: Я говорил про мониторинг. Это один из критериев мониторинга на уровне приложения. Мы смотрим количество кликов... Допустим, у нас есть площадка, мы знаем, какой у нее CTR. Если мы видим, что у нас в какой-то момент CTR резко возрастает (количество кликов возрастает), то для нас это повод разбираться, что там происходит.

Можно еще смотреть на статистику и отслеживать динамику. Либо запустилась какая-то мощная рекламная кампания, и она действительно приносит такие клики, либо это какой-то паразитный трафик, который надо отслеживать. В любом случае, информация о кликах накапливается, она хранится какое-то время. Если возникает такая ситуация, потом это можно просто разобрать, вычистить и подкрутить статистику.

Вопрос из зала: (Неразборчиво, без микрофона).

Артем Вольфтруб: Были, конечно. Есть такие критерии, которые отслеживают динамику относительно данных за период. Отслеживают среднее количество кликов за период, скажем, по сайту или даже по рекламному месту. Если оно резко возрастает, то это, конечно...

Ведущий: У нас закончилось время, к сожалению.

Node.JS

Андрей Костенко

node.js

Андрей Костенко: Перейдем к первому слайду.

УГОЛОК КЭПА

Пример: счетчик посещений

- Получаем http запрос
- Смотрим, есть ли пользователь с этой сессией
- Если нужно – добавляем пользователя
- Добавляем запись “Вася зашел на index.html”

Сначала я побуду Капитаном Очевидностью, расскажу, зачем вообще нужно использовать node.js, принципы его работы, чем он отличается от всего остального.

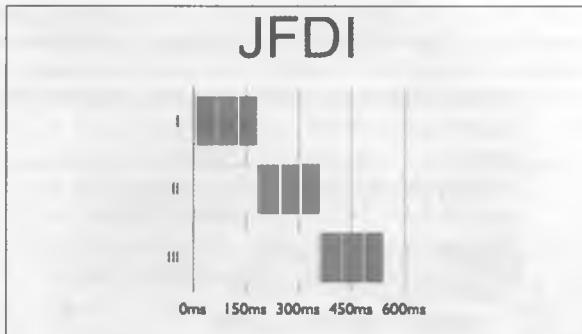
Начну с такого простого примера: нужно реализовать обычный счетчик посещений. Мы получаем http-запрос. Смотрим, есть ли пользователь с таким IP'шником или сессией, неважно. Если нет, добавляем пользователя. В любом случае добавляем запись: «Вася зашел на index.html». Кажется, простая задача.

Варианты решения

JFDI

- Цикл
- SELECT * FROM users ...
- INSERT INTO users
- INSERT INTO stats

Ее можно решить несколькими вариантами. Первый вариант, самый простой — получаем запрос, делаем SELECT-INSERT-INSERT. Стандартная ситуация.



Но если мы посмотрим на log запросов, то, очевидно, что, во-первых, следующий пользователь будет ждать, пока обработается предыдущий.

Но даже если мы посмотрим на обработку запроса (это я в качестве примера показал), зеленым мы будем видеть те части, где наш процессор занимается полезным делом. Он процессит http-запрос. Дальше большую часть времени валяет дурака, дожидается, когда выполнится запрос к базе данных, опять что-то делает чуть-чуть, валяет дурака и так далее. 90% времени процессор занимается тем, что валяет дурака.

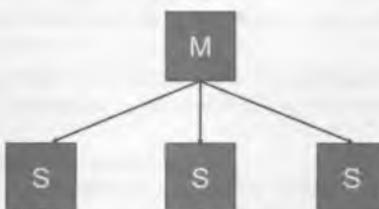
То же самое с каждым запросом. Этот код, вне зависимости от языка реализации, от того, оптимально или неоптимально это будет работать, если не смотреть на базу данных, в любом случае он будет феерически тупить, хоть на С его переписать, хоть как оптимизировать.

Скорость операций

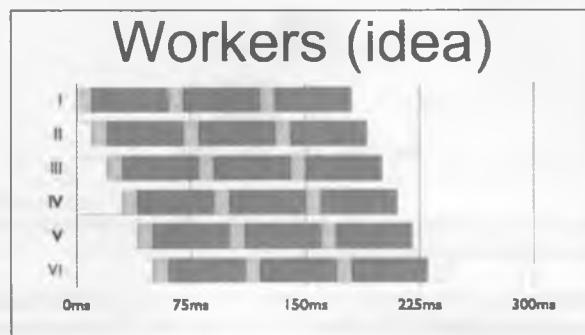
- CPU tick - 1
- CPU Cache access - X
- Context switch - XX
- Memory access - XXX
- Disk access - XXXX
- Network access - XXXXXX

Варианты решения

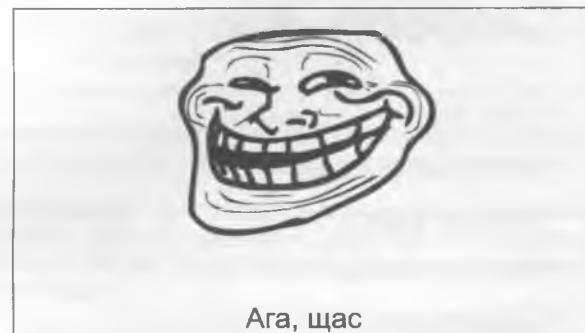
Workers



Следующий вариант, это тоже стандартный способ, которым пользуется много приложений. Это использование worker'ов. С worker'ами процесс получает задачу, master получает задачу, отдает задачу worker'ам. Worker'ы, в свою очередь, в несколько потоков или процессов выполняют эту задачу. Так работает тот же Apache, в качестве самого популярного примера.



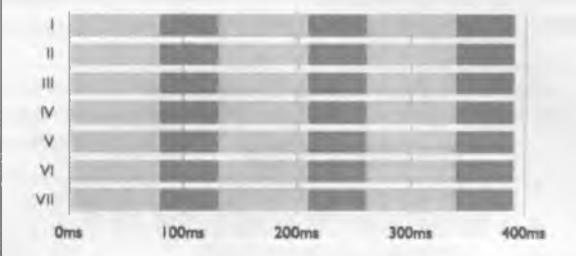
Как это работает в самом идеальном красивом случае. Первый worker работает, выполняет какую-то часть задачи. Идет по сети, например, выполнить SQL-запрос. Дальше, CPU не используется. В это время планировщик задачи отдает процессорное время второму процессу. Работает второй процесс. Соответственно, время выполнения задач, конечно же, существенно ниже в идеальном случае. Но, так как наша жизнь не совсем идеальная, то...



Надеюсь, это не попадает под NDA.

Есть уровни террористической угрозы во всем мире — меньше, больше. В Rambler'e была такая критическая ситуация, называлась «сиськи на главной». Это значит, что появилась мегапопулярная новость, на которую приходит очень много трафика. Нагрузка сразу вырастает вдвое, пиково. Еще кое-как справляется. Тут происходит такая ситуация — сбрасывается кэш.

Workers (real world)



Все пользователи рвутся в один момент времени использовать... Все запросы начинаются в один момент времени. Это самая клиническая ситуация. Наполовину она высосана из пальца.

Но, получается, так как планировщик задач в операционной системе не знает, когда правильно раздать процессорное время, то получается так. Сначала все дружно рвутся использовать процессорное время, вырывая процессорное время друг у друга, переключая контексты. Потом мы все дружно отправляем SQL-запрос — и опять начинается валяние дурака.

Происходит это почему. Примерно как в московских пробках. Если бы все в московских пробках обезжали препятствия по очереди, по-правильному, то московские пробки были бы намного меньше. Но все же начинают бибикать, обгонять друг друга, выруливать по обочине. В результате получается то, что получается.

То же самое примерно работает с Apache'ми, с другими сервисами, которые используют либо множество потоков, либо множество treadов. Процессор используется не на всю, а большая часть процессорного времени тратится на ерунду всякую.

Workers

- Работает быстрее
- Вариант I несложно превратить в II (в большинстве случаев)
- Поддержка нескольких CPU

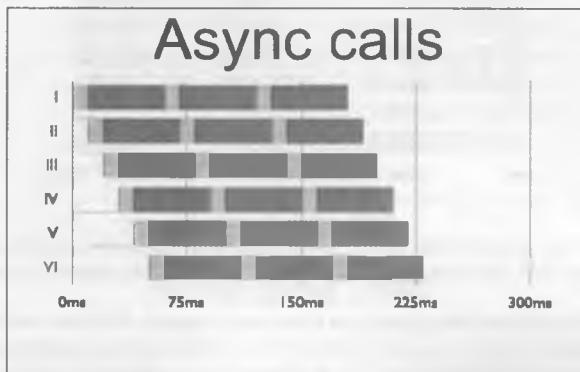
Плюсы работы с worker'ами.

- Понятно, что worker работает быстрее, чем один поток, это очевидно. Работает намного быстрее, чем один поток, и на них можно что-то нормально писать.
- Первый вариант, если вы написали какой-то сервер «на коленке», который использует while/accept, то его несложно превратить во второй вариант, добавить туда fork'ов или еще чего-то. Практически без изменений кода, если код примерно нормальный, получится второй вариант.
- Еще один плюс, о котором чуть позже, это поддержка нескольких CPU. Это касается большинства новых серверов, поэтому это существенный плюс.

О недостатках (они упали вниз на слайде).

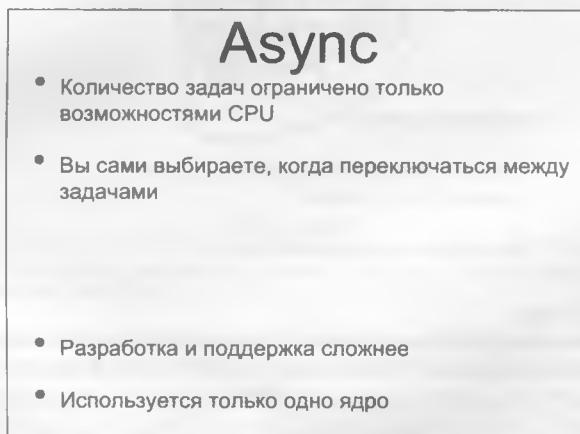
- Если много worker'ов, и используются fork'и, то мы наталкиваемся на другое ограничение. Мы уходим в минус по оперативной памяти. Кушается вся наша оперативная память. Например, запустить 10 тысяч процессов — уже как-то не то. Иногда такое нужно.

- В случае с тредами получается немного другая проблема. Во-первых, во многих популярных языках программирования они сделаны через пятую точку. Там, где они сделаны нормально, нужно возиться с shared-переменными и так далее. В CPU тоже нагрузка не будет использоваться по максимуму.



Мы перейдем к асинхронным вызовам. Для этого есть умное слово — «мультиплексирование». Но будем применять слово «асинхронное». Как они работают.

Мы начинаем выполнять запрос. В тот момент, когда мы переходим к вызову `select'a`, мы не делаем вызов, `select` из таблицы и дожидаемся ответа. Мы говорим: «Начать `select`» и смотрим, что мы в это время можем сделать другого. У нас пришел другой запрос. Мы начинаем обрабатывать его. Опять начинаем `select`, опять смотрим, что мы можем делать. В этом случае мы сами решаем, когда и для чего нам нужно использовать процессорное время.



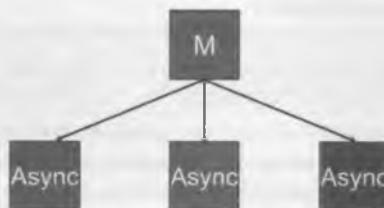
Вот, соответственно, плюсы и недостатки асинхронных вызовов.

- Плюс в том, что, используя асинхронный Framework, вы можете использовать ваш CPU действительно на 100%. По сравнению с worker'ами, даже для многих задач асинхронная работа на одном ядре получится намного быстрее, чем на двух-четырех (для некоторых задач).
- Вы сами выбираете, когда переключаться между задачами. Соответственно, за счет этого еще будет вам прирост производительности.

Недостатки.

- Разработка и поддержка сложнее (опять расскажу почему), чем в случае с worker'ами. Не получится всунуть три fork'a, добавить проверку, чтобы fork'ов не было 10 тысяч, и все.
- Еще из недостатков. В этом случае я покажу, почему. Будет использоваться только одно ядро.
- Нельзя быть чуть-чуть мультиплексирующим приложением. Можно, но не нужно. Если вы пользуетесь этим вариантом, то все модули, вся работа с сетью, с диском должна быть неблокирующей.

Workers + async



Есть вот такой компромиссный вариант. Когда у нас есть master, и есть несколько асинхронных worker'ов, которые работают примерно так же, как я показал на предыдущем слайде.

Workers + Async

- Вы используете все возможности вашего сервера
- Вы сами выбираете, когда переключаться между задачами
- Разработка и поддержка кода становится еще сложнее

В таком случае мы получаем и плюсы использования нескольких процессоров, и все плюсы асинхронного приложения. Из недостатков опять же писать букв придется еще больше.

Какой вариант лучше?

Не знаю

Какой вариант лучше подходит для вашей задачи, я не знаю, и никто не знает.

QA

- cron
- while/accept
- forks
- async

Но для себя я сделал такое мини-правило.

- Если задачу можно сделать с помощью cron'a, я ее засовываю в cron.
- Если задачу можно сделать самым простым способом, while/accept'ом, и в ближайшем будущем не понадобится ее расширять, то использую второй вариант.
- В третьем случае использую fork, потому что он еще проще в разработке и поддержке.
- Но в случае, если все-таки нагрузка большая, и у вас есть сервер, где очень много соединений, и вам нужно их все обрабатывать, то используется асинхронное приложение.

Async

Multiplexing

Socket I/O

- fcntl (O_ASYNC|O_NONBLOCK)
- read/write (EAGAIN)

Сейчас расскажу еще всякой теоретической лабуды о том, как это вообще работает.

Во-первых, для socket'ов используется опция O_NONBLOCK, которая говорит после read или write. Обычно, если мы читаем что-то из read'a, то мы дожидаемся п-ного количества байтов и выходим. В этом случае read или write, если будет блокироваться, он возвращает ошибку EAGAIN. Это значит, что он ничего не прочитал, но и не заблокировался.

Через некоторое время, если мы еще раз дернем read, то мы получим наши данные. Но если у нас один socket, мы можем через некоторое время дернуть read. Если у нас, например, 10 тысяч socket'ов, и нужно узнать, пришли ли туда данные.

Работа с сетью

- select/poll $O(N=total\ fd)$
- kqueue (BSD only) $O(N=active\ fd)$
- epoll (linux only) $O(N=active\ fd)$

Для этого случая используются вот такие функции.

- Select/poll работает везде. Несмотря на то что он работает везде, и вы видите еще две строчки, то, наверное, он недостаточно хорошо работает везде. Так и есть — он работает очень медленно. При увеличении количества descriptor'ов, за которыми мы хотим следить, скорость его работы существенно падает. Падает линейно. Если у вас будет 10, 20 тысяч одновременных соединений, она падает чуть ли не до нуля и становится неизабельной.
- Так как Линус Торвальдс еще не купил FreeBSD, то мы получаем два разных вызова, которые делают примерно одно и то же. Epoll в Linux'е и kqueue во FreeBSD. Их скорость зависит только от количества активных descriptor'ов, то есть только от количества descriptor'ов, которое пришло. Их производительность существенно выше и позволяет обрабатывать довольно большое количество, одновременно следить за большим количеством descriptor'ов.

Таймеры

- setitimer
- alarm

Таймеры. Есть функции для работы с таймером.

- Setitimer — это для своего кода.
- Alarm — для кода, которому вы хотите сделать нехороший сюрприз.

Абстракция от бардака

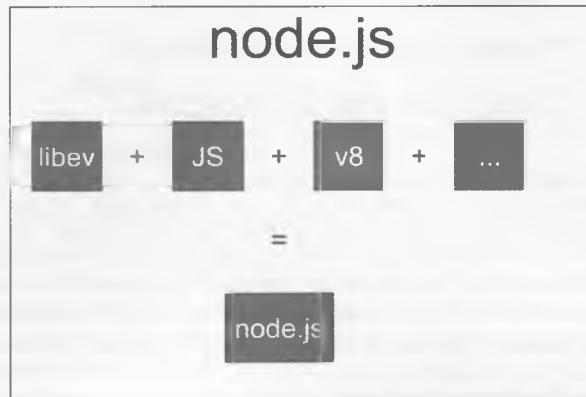
- libev
- libevent

Для одной операционной системы у нас kqueue, для другой — epoll. Куча бардака с таймерами и так далее.

Должна быть еще одна система, которая следит там, где я говорил: «А сейчас мы посмотрим, чем бы нам заняться еще». Эта система называется Event loop. У нее есть список задач, и она

выбирает, чем из задач нужно заняться. Эти библиотеки, во-первых, являются абстракцией от кучи системных вызовов. Они работают на уровне callback'ов. Они предоставляют один Event loop.

Event loop должен быть один. Можно, конечно, поизвращаться с несколькими. Но в нормальном варианте он должен быть один. Разница между ними, которую я знаю, это то, что libev быстрее, чем libevent. Опять же знаю со слуха. Именно тестирование я не проводил.



Все-таки перейдем к теме. Что такое node.js. Это продукт, который комбинирует вот эту библиотеку libevent, которая существенно быстрее конкурентов. Берет движок JavaScript V8 от Google, который делает довольно многие современные интерпретируемые языки программирования по производительности, и, соответственно, берет язык программирования JavaScript, который уже предназначен для неблокирующей работы.

Когда вы пишете свою домашнюю страничку на index.html, добавляете туда JavaScript, не блокирующие приложения, вы уже пишете асинхронное приложение. Соответственно, он максимально близок к этой разработке. Его использовать намного удобнее. Даже, насколько я помню, в версии Firefox 3.3.0 весь JavaScript выполнялся в одном потоке, в одном процессе. Если какой-то из процессов завешивал Firefox, то зависало все. Не выполнялось ничего из других вкладок.

Что в коробке?

- HTTP
- FS
- Прозрачный SSL
- DNS
- `setTimeout`, `setInterval`

Что у нас есть с node.js. Вы поставили node.js. У вас сразу есть работа с HTTP, сразу работа с файловой системой неблокирующая. Сразу SSL, DNS, таймеры. Об этом вы не задумывались.

тесь даже. Это у вас просто есть. Вы работаете, как с JavaScript'ом так же, как вы работаете с AJAX'ом на странице.

Что еще?

- npm = cpan = pip = emerge
- ndb
- Socket.IO
- CoffeeScript
- async-интерфейс к 99% приложений, с которыми вам придется работать*

* По данным анализа Rage & Koryta Inc. на конец 2011-го года

Чего нет, но что полезного даст вам node.js при наличии желания.

- NPM — это Node Package Manager, с помощью которого можно поставить практически любой модуль, который ставится, у которого не файлятся тесты. У node.js есть Debugger. О нем чуть-чуть поподробнее, потому что с debugging'ом тут не все так просто, не все так хорошо.
- Есть модуль Socket.IO, который предоставляет абстракцию. Вы можете написать один код, который будет работать и на сервере с сетью, и на клиенте в браузере он точно так же будет работать с socket'ами, используя какой-то из возможных протоколов. Если поддерживается WebSocket, то WebSocket. Если не поддерживается WebSocket, то WebSocket не поддерживается. Используется либо flash, либо костыли с AJAX'ами.
- CoffeeScript — это больше для любителей извращения. Это специальный язык программирования, который компилируется в JavaScript. Но при этом разработка становится чуть удобнее, потому что те костыли, которые вбиваются JavaScript'ом, не вошли в стандарт, но их часто приходится использовать.

Например, я часто пишу «var self = this», чтобы образовалось замыкание. Там это делать не нужно. Там есть маленькие плюсы. Необъявленная переменна без var'a все равно локальная. Так же, как в Python'e. Не объявляется глобальная переменная, если не писать var.

- Async-интерфейс к 99% приложений. По собственному опыту асинхронный интерфейс был ко всему, к чему было нужно. PostgreSQL, MySQL, всякие Beanstalk'и и так далее. Есть интерфейс ко всему, и он будет неблокирующим — вы сразу можете его использовать.

Особенности

- throw Exception
- debugger
- profiler
- spaghetti

Опять, в fail-конвертации в PowerPoint — не конвертируйте Keynote в PowerPoint. Вот такое получается.

Во-первых, на какие проблемы натыкается человек, когда пытается (по себе) поработать с node.js. Во-первых, с throw Exception, с обычным приложением на Perl'е, на Python'е — без проблем. Бросили Exception, выловили его на более высоком уровне. Отлично. Как-то обработали эту ошибку или спрятали подальше от менеджеров.

Здесь так не получится. Вы выбросили исключение, а дальше обычно приложение выходит по стеку вызовов наверх, а у вас его нет. У вас, так как приложение асинхронное, оно просто не знает, что ему делать дальше. Поэтому выбрасывание исключения — только в том случае, если вы хотите аварийно завершить приложение. Либо используйте ту часть, где не используются callback'и.

Debugger — то же самое. Есть Firebug Debugger — аналогия та же самая: вы не можете войти в onClick или еще что-то. Есть свои особенности работы с callback'ами. Не получится войти внутрь функции, которую требует callback. То же самое с профайлерами, опять же из-за отсутствия call stack'a.

Еще одна проблема. Иногда, когда разрабатывается код, есть такая проблема. Хочется написать что-то побольше и потом из этого выковыривать функции, то есть выделять логические блоки, функции и так далее. Это есть такой большой кусок кода.

С node.js другая проблема. Эти спагетти, если не делается... Оно предоставляет очень хороший интерфейс для делания спагетти, для использования замыкания из верхнего уровня и так далее. Если написан код с кучей callback'ов и анонимных функций, назад его разложить получается иногда очень сложно.

Success stories

- ВКонтакте XMPP
- 37Signals
- Wikia
- Palm/HP WebOS

Success stories — это из тех, которые я знаю, и большая часть из тех, которые написана на node.js Success stories. Но первый случай — это тот, который работает, тот, которым мы все даже пользуемся, он тоже работает. Это ВКонтакте XMPP. Когда-то они даже обещали открыть исходники его. Но пока нет. Используют node.js 37Signals, Wikia, Palm/HP WebOS. Использовали его и в Rambler'e.

Тесты производительности

*

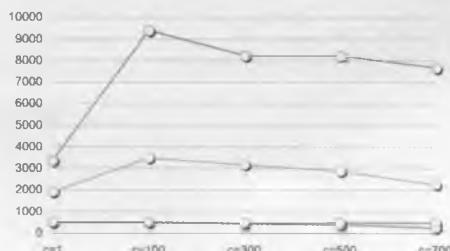
* Эти тесты тестируют неверно,
результаты неправильные
и пользы от них никакой

Давайте перейдем к тестам производительности. Я сделал вот такую пометку, потому что, действительно, тесты производительности значат очень мало. Так же, как с тем, если вы, допустим, используете сортировку пузырьком и сортировку, например, QuickSort. На медленном компьютере QuickSort будет работать медленнее, чем сортировка пузырьком на очень быстром.

Но при увеличении количества данных в любом случае, какой бы медленный компьютер ни был с QuickSort'ом, и какой бы быстрый компьютер с пузырьком ни был, все равно медленный компьютер с QuickSort'ом уделает быстрый, потому что там используется более быстрый алгоритм.

Бесполезный график

—> AE:Loop —> AE (EV) — Twisted —> node.js



Эти данные не показывают ничего. Они показывают, что в 22 часа по восточно-американскому времени на моем компьютере HTTP-сервер обрабатывал такое количество request'ов в секунду при таком количестве одновременных подключений. Не значит они ничего, потому что при увеличении количества соединений скорость практически не будет падать.

Здесь тестируется просто деталь реализации языков, реализации модуля работы с HTTP. Здесь тестируется, по большей части, сам язык программирования. Неважно, что из этого использовать, потому что для этого можно доставить два сервера, четыре сервера. Если используется грамотная технология, то... Здесь важна именно технология, а не скорость работы. Но Node.js всех делает все равно.



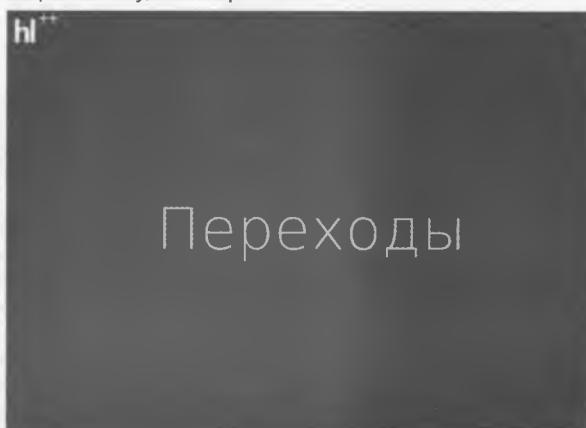
Мужской голос: Большое спасибо. Вас ждет кофе-брейк и розыгрыш призов. Вопросы, к сожалению, не успеваем — в кулуарах. Большое спасибо.

AJAX Layout

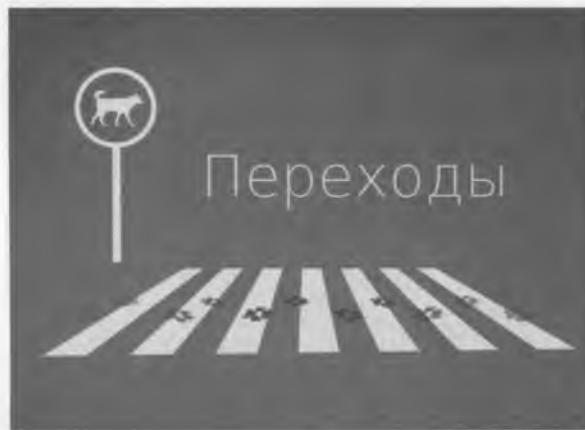
Олег Илларионов



Олег Илларионов: Рад здесь всех видеть. Сегодня хотелось бы рассказать про очень интересную вещь. Про то, каким веб стал сейчас, как он изменился и к чему он пришел. Мы называем это Ajax Layout. Я не знаю, может быть, кто-то использовал это словосочетание до нас. Может быть, нет. Но мы не нашли термина, как назвать эту сущность. Разные люди называют это по-разному. Сейчас расскажу, о чём речь.



Начать хотелось бы с переходов.

hl⁺⁺

- 1) Пользователь кликает
- 2) Индикатор загрузки
- 3) Получение контента
- 4) Получение статики
- 5) Отображение контента
- 6) Изменение адреса

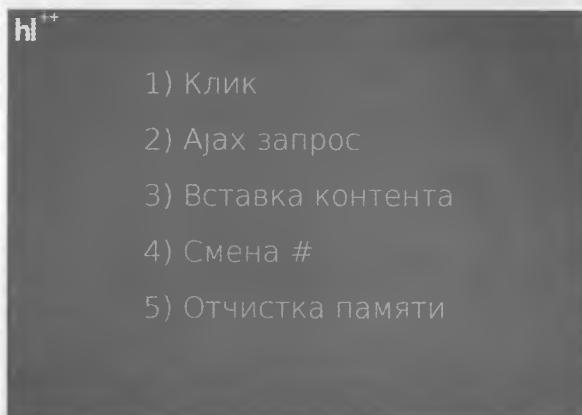
Что такое переходы. Когда пользователь переходит с одной страницы на другую. Раньше всю ответственность за переходы брал на себя браузер. Браузер сам ловил событие клика по ссылке. Браузер сам показывал индикатор загрузки на странице. Он получал контент. Потом он смотрел, какие стили и какие скрипты требует эта страница. Получал стили и скрипты.

По мере того как он получил контент, он рисовал последовательно контент на определенной странице. Потом он отображал его и менял адрес в заголовке. Большинство браузеров так и работает. Большинство сайтов тоже так выглядят. Вы прекрасно знаете об этом.



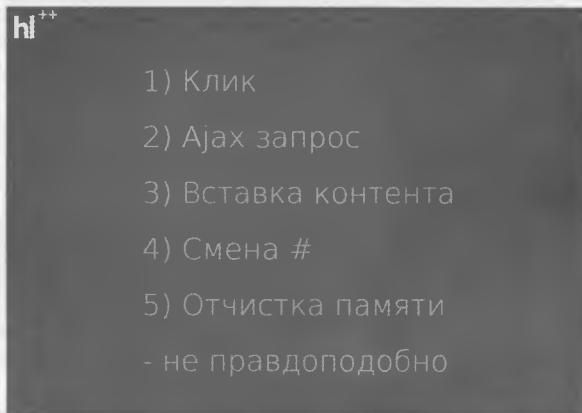
Но в определенный момент все поняли, что такая концепция недостаточно хорошо работает. Что это не дает возможности оптимизировать очень многие вещи. Это не дает возможности контролировать многие процессы, потому что у нас полностью перезагружается страница, мы все теряем и все получаем заново.

Мы должны полностью формировать страницу с того момента, с которого она была. Мы не можем хранить события, мы не можем показывать что-то. Нам тяжело работать с Long Poll'ом. Веб очень простой, не Real Time'овый. Нужно это изменить. Начали экспериментировать. Многие сайты реализовали в разной степени прогресса эту технологию.



Началось это с простой схемы, когда пользователь кликает. JavaScript отправляет Ajax-запрос на какой-то контент. После этого контент вставляется в страницу, и устанавливается хеш в браузере. После чего очищается память, какие-то локальные переменные, которые были установлены, чтобы не засорять память.

Эта схема работала недостаточно хорошо. Во-первых, адреса были некрасивыми. Были хеши, которые передавались на серверную сторону. Это была основная их причина. Если мы имеем страницу с хешом, то хеш не отправится к серверу, и нам нужно отловить его на JS и отправить самим. Это значит, что когда пользователь вставляет к себе в браузер ссылку а-ля «vkontakte.ru/hash что-то там», мы должны сначала загрузить какую-то страницу, потом сделать Ajax-запрос на сервер, поймав то, что у нас в адресе. После этого мы можем ее как-то отображать и работать дальше. Это плохо.



hl⁺⁺

- 1) Клик
- 2) Контент 2) Статика
- 3) Отображение контента
- 4) vk.com/#blah ?

После чего был следующий этап, как это все может работать. Это момент, когда мы делаем клик, очень хорошо оптимизируем скорость нагрузки за счет того, что мы заранее знаем, для каких страниц какие стили и скрипты нужны. Мы просто храним в config'e определенную схему сайта, что у нас есть такие-то страницы, которые требуют такие-то скрипты по «регуляркам». У нас есть «регулярки», которые описывают страницы. Мы находим нужную «регулярку» и решаем, какие скрипты нужно подключить. Тем самым мы начинаем загружать статику не в тот момент, когда мы уже получили контент, а одновременно. Тем самым, если статика не закэширована, мы ускоряем загрузку страницы. После чего мы отображаем контент и меняем location bar. Но есть проблемы с адресом, поэтому это плохо.

Еще один момент — то, что нам приходится изображать загрузку страницы. Когда мы делаем Ajax-запрос, в большинстве браузеров ничего не крутится. Пользователь думает, что запроса нет, это очень плохо. Поэтому вначале мы использовали анимированный favicon. Просто favicon, который как бы крутится. Это простой способ изобразить что-то.

hl⁺⁺

- 1) Клик
- 2) Контент 2) Статика
- 3) Отображение контента
- 4) History Api

Но потом мы пришли к History API, и он позволил избавиться от хешей в тех браузерах, которые это поддерживают. Естественно, в браузерах, которые этого не поддерживают, мы не можем работать с History API. Когда была выкачана первая версия всего этого дела (это было, наверное, где-то в ноябре, даже чуть попозже, прошлого года), это работало только в Chrome. На тот момент был всего 1 браузер, который поддерживал History API. Все остальные работали с хешами.

hl⁺⁺

- 1) Клик
- 2) Контенет 2) Статика
- 3) Отображение контента
- 4) History Api

History API стоит выделить, потому что он, на самом деле, дал такой шаг. Если бы его не было, у нас просто не было бы надежды, и мы не стали бы этого делать. Мы бы оставили старую схему и пытались бы искать какие-то другие способы — каждый раз перезагружать всякие notify, чаты и так далее.

hl⁺⁺

- 1) Клик
- 2) Анимированный favicon
- 3) Контенет 3) Статика
- 4) Отображение контента
- 5) vk.com/blah, # и хаки
- правдоподобно

Что у нас осталось. Мы отображаем контент. Используем нормальный адрес там, где можно. Где нет — мы используем решеточку (#) и используем разные хаки. Например, известно, что для того, чтобы старые версии IE отловили смену события, поняли, что у нас страница перешла с одной на другую, нам нужно использовать iframe. Нужно менять там страницу.

В противном случае просто не будет работать навигация. Это очень неприятно для пользователя, когда он не может нажать на кнопку «назад» и не может вернуться на предыдущую страницу. Это работало достаточно правдоподобно. Это уже увидели пользователи.

hl⁺⁺

- 1) Клик
- 2) Анимированный favicon
- 3) Контенет 3) Статика
- 4) Отображение контента
- 5) vk.com/blah, # и хаки
- но визуально медленно

Следующий момент заключается в том, что это работало медленно. Казалось бы, время генерации на сервере только ускорилось, потому что нам не нужно получать левое меню, footer, еще что-то. Понятно, что это хорошо кэшируется, но это трафик. Нам не нужно его получать. Это проблема, потому что страница вроде бы генерируется быстрее, а визуально это медленнее.

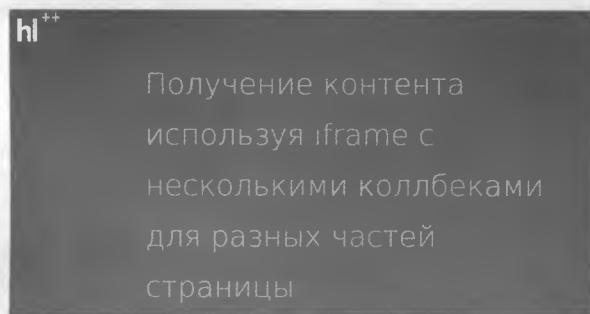
Но очень быстро стало понятно, в чем дело. Все дело в том, что страница отображается только в тот момент, когда мы получили весь контент. Мы получили гигантскую страницу и только потом стали ее отрисовывать. В то время как при классической загрузке браузер получил первую часть. Если у него есть нужные скрипты и нужные стили (если он их уже получил), он отрисует это и по мере получения информации будет отрисовывать.



Поэтому пришлось осознать эту проблему. Понять, что вся проблема в контенте и вернуться к каким-то истокам.



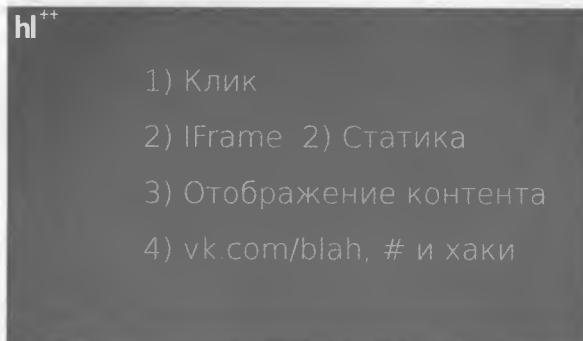
Все начать заново осмыслять. Стало ясно, что Ajax не подходит для решения этой задачи.



Поэтому пришлось использовать iframe. У нас есть iframe, и когда мы хотим перейти на другую страницу, мы открываем скрытый iframe, в котором есть части. Части — это разные части страницы. Вместо того чтобы брать и загружать всю страницу Ajax'ом сразу, мы загружаем ее по частям.

Сначала header... В этом iframe'е есть функция, которая перекидывает наверх код в header. Потом еще что-то, еще что-то, еще что-то. Все это через iframe. Все это кажется нереальным. Сначала мы думали, что это не будет работать. Как это? Это не может быть быстрее.

Но потом мы увидели, что это быстрее в разы. Пользователь плавно получает контент, плавно его видит, и у него остается ощущение скорости. Он думает, что страница быстрее. Это как раз то, на чем многие сайты остановились. Они не пошли по этому пути и, мне кажется, сделали большую ошибку.

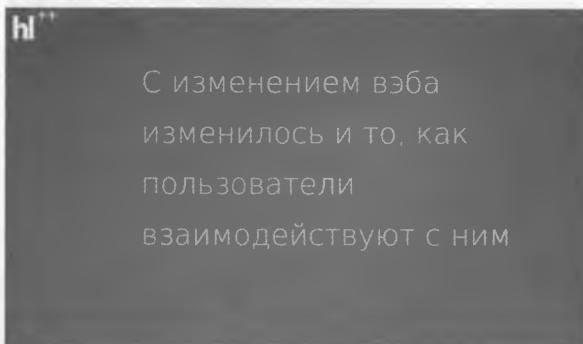


Таким образом, все стало работать хорошо.



Но мы столкнулись со следующей проблемой. Естественно, все это затевалось ради Long Poll'a, ради Real Time'a, чтобы мы могли получать какие-то события. Но только представьте, какие нагрузки будут, если мы просто берем и начинаем держать Long Poll. Вот сайт, у него немаленькая посещаемость и вдруг ни с того ни с сего нужно держать по коннекту на каждого пользователя постоянно. Но это еще полпроблемы. Это проблема решаемая.

Но у каждого пользователя по 20 открытых вкладок. Это уже, действительно, проблема. Это даже целых две проблемы.





С изменением вэба
изменилось и то, как
пользователи
взаимодействуют с ним.

#realtime



1) Постоянный long poll

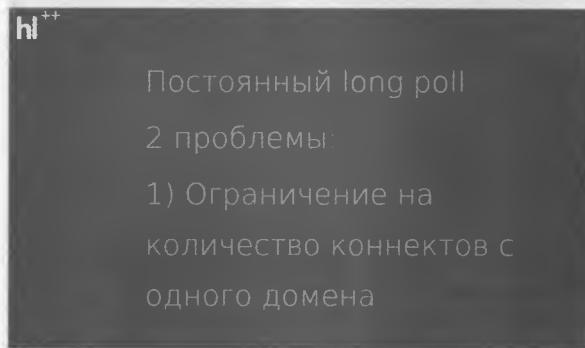


1) Постоянный long poll
2) Проблемы?

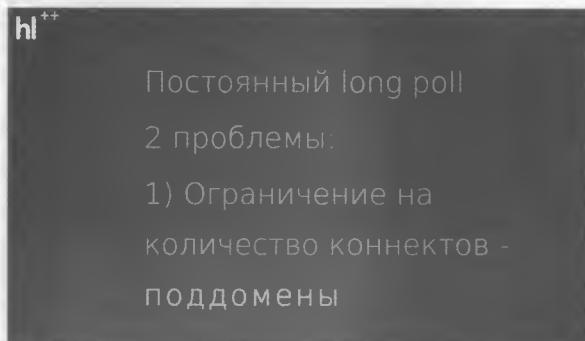


Постоянный long poll
2 проблемы:

Это целых две проблемы. Первая проблема в том, что мы не можем держать много коннектов на один домен. Мы не можем для каждой вкладки... Браузеры это ограничивают. Но эта проблема легко решается.

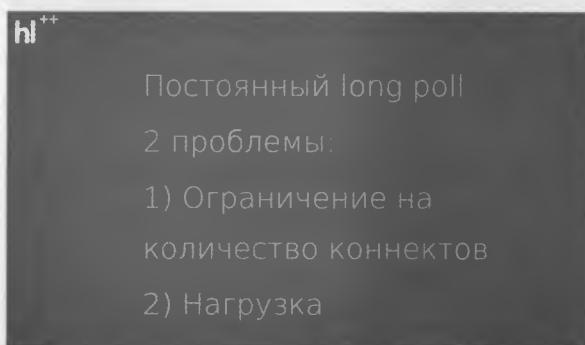


Мы просто берем и делаем поддомены.



Так делал «Facebook». Не смотрел, как они сейчас делают. Скорее всего, они тоже все переосмыслили, но я не уверен. Может быть, они до сих пор на этом состоянии.

Много поддоменов. Для каждой вкладки свой поддомен — все ok, работает.



Но нагрузка. Для каждой вкладки свой long poll — это неприемлемо.

hl⁺⁺

Постоянный long poll
Нагрузка ++

hl⁺⁺

Нагрузка ++
Хочется общаться между
ВКладками

Появилась такая мечта, что нужно общаться между вкладками. У нас много вкладок (браузер, вкладки), и хочется общаться между ними. Хочется иметь возможность из одной вкладки сказать, что «я буду держать long poll и передавать вам всем, что там происходит, а вы просто живите себе и ничего не делайте».

Сначала казалось, что это только flash.

hl⁺⁺

Нагрузка ++
Хочется общаться между
ВКладками

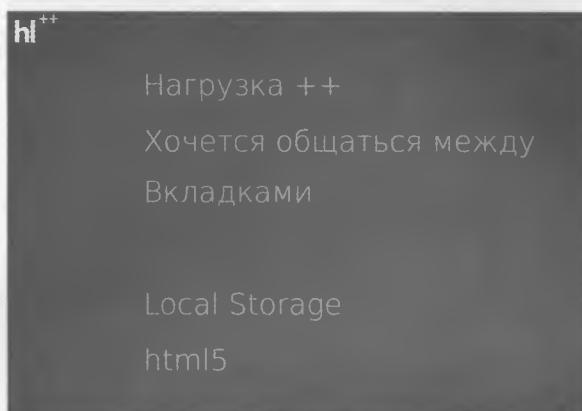
Local Connection

hl⁺⁺

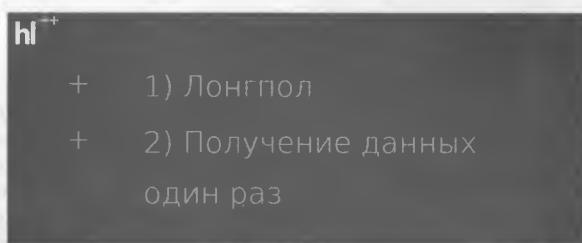
Нагрузка ++
Хочется общаться между
ВКладками

Local Connection
Flash

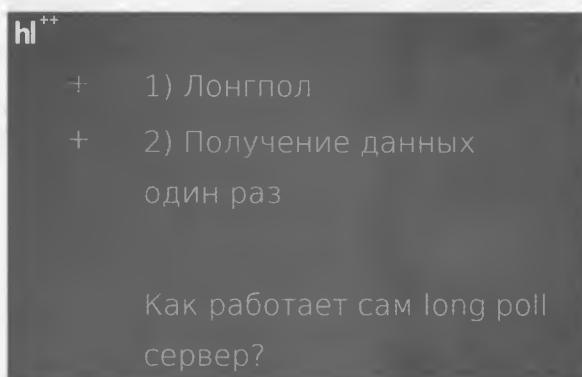
Local Connection, Flash, и нет другого выхода. Это очень печально, потому что завязывать сайт на flash'e, от которого очень хочется отказаться, потому что он где-то вылетает... Все знают проблемы с разными версиями flash-player'a и с тем, что там происходит.



Но потом был открыт html5. Многие, наверное, в курсе, что есть такая вещь, как web-storage.

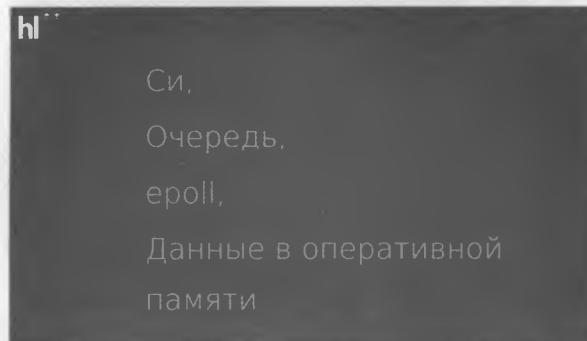


Web-storage — это стандартный html5, который позволяет нам хранить какую-то информацию прямо в браузере. Это никак не помогло бы нам, если бы не было события window.onstorage. Мы можем в момент положения каких-то данных в local storage отловить это событие, посмотреть, что туда положили и как на это отреагировали. То есть мы можем общаться между вкладками, причем без flash'a и в достаточно большом количестве браузеров. Можно сказать, что во всех современных браузерах, более или менее используемых, даже в мобильных. Единственное — Opera Mini, не знаю, насколько на это поддерживает. Но для нее у нас отключены Ajax-переходы, поэтому живем.



Таким образом, у нас появился long poll сервер, который держит один коннект, вкладки общаются, решают, кто будет держать. Если он очень долго не отвечает (то есть вкладка была закрыта), кто-то другой берет эту задачу и решает ее.

Хотелось бы рассказать, как устроен сам long poll сервер. Мне известно, что многие тоже решают эту проблему, многие используют Real Time в своих проектах и пытаются организовать что-то подобное.



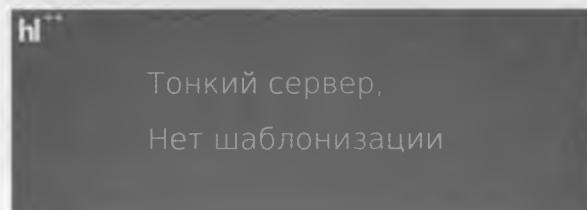
Понятно, что с нашими нагрузками это возможно только на С и на низком уровне (очередь, все дела, никакой работы с диском). Только какие-то локальные вещи, только оперативная память.

Для других проектов это все можно делать совершенно по-другому. Но в нашем случае только так. Самое главное — это то, что ни в коем случае на сторону long poll'a не нужно навешивать всякой логики. Возможно, изначально может показаться, что long poll сервер должен быть таким большим сервером, которым узнает о каком-то событии, генерирует html, получает long pack, проверяет приватность/неприватность и только потом что-то отдает пользователю.

Это неправильно, потому что в условиях, когда нужно держать висящие коннекты в оперативке, это будет медленно работать. Это очень сложно реализовать.



Поэтому такой long poll сервер должен быть очень тонким. Он не должен ничего делать, кроме как передавать события и хранить, пока есть память, некая очередь. Мы называем это Q-Engine.



hi⁺⁺

Тонкий сервер,
Нет шаблонизации,
Нет бизнеслогики

Нет шаблонизации. Нет бизнес-логики.

hi⁺⁺

Тонкий сервер,

Логика, шаблонизация,
Лангпак - во время
генерации события.

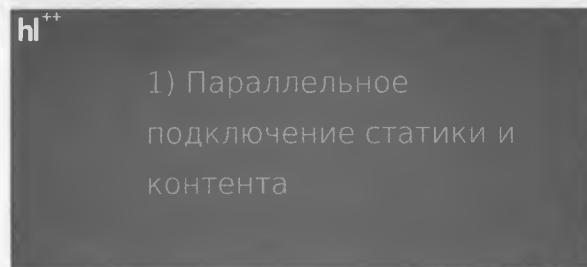
Нет ничего. Только передача событий.

Скорость

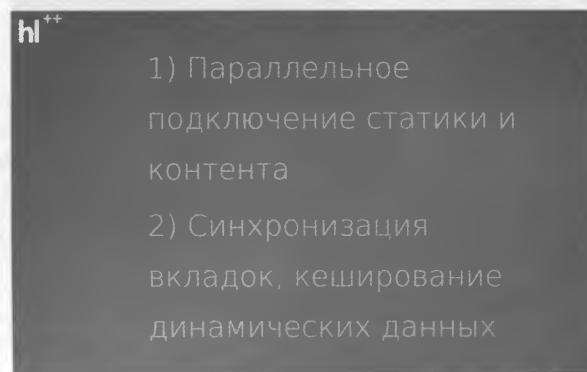
Это работает быстро.



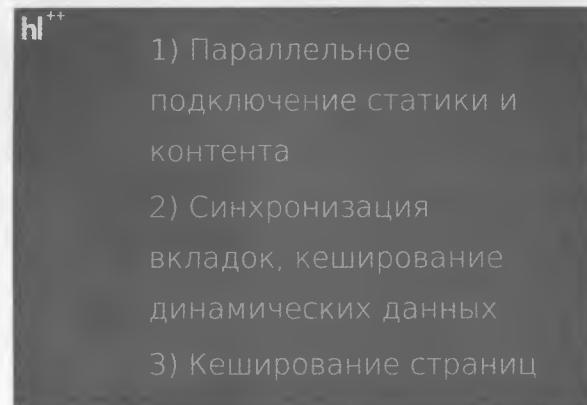
Последняя, очень важная тема, о которой хотелось бы рассказать, это бенефиты, которые от всего этого можно получить. Что вообще это дает в плане производительности, в плане борьбы с нагрузкой.



Первое — это подключение статики и контента, о чем я уже сказал.

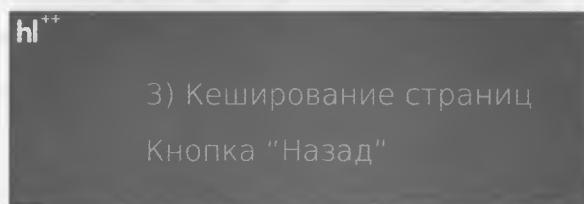


Вторая вещь — это синхронизация вкладок и кэширование динамических данных. Мы можем в local storage поместить что-то класть, что мы хотим получить на каждой вкладке. Например, онлайн-чат, который можно включить (чертенький такой), друзья. В общем, многие вещи можно положить в local storage и там с ними работать.



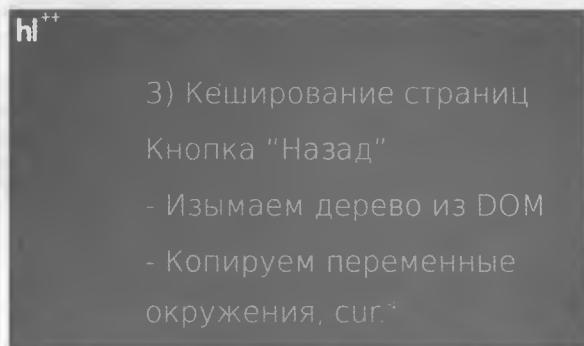
Еще один момент — это кэширование страниц. Когда мы все делаем на клиентской стороне, мы можем не делать некоторых запросов, и пользователь будет думать, что они происходят,

что он перешел на другую страницу. На самом деле, ничего не происходит. Мы просто, имея информацию, которая у нас уже есть, отобразили ему другую страницу и решили таким образом проблему, не делая запроса к серверу и снижая нагрузки.



В первую очередь, это касается кнопок «Назад». Когда мы перешли с одной страницы на другую, а потом нажали «назад» в браузере. Казалось быть, что сделает большинство браузеров. Они сделают запрос к серверу и загрузят заново страницу. Но зачем это делать, если пользователь уже был на этой странице, и мы можем просто показать эту страницу.

Возникает вопрос: «Как это реализовать?». Как сделать так, чтобы мы могли показать какую-то страницу, на которой он был, у нее была верстка, там выполнялись какие-то скрипты. Очень много вещей, которые в окружении. Как их всех заставить работать?

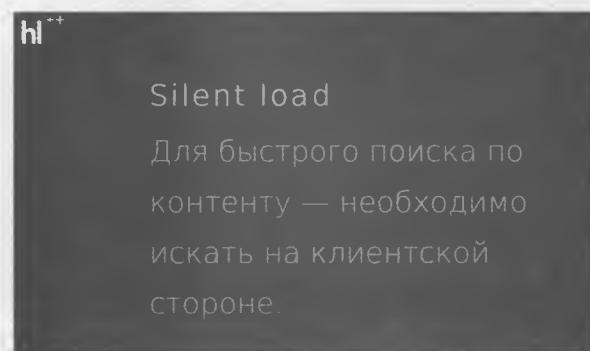


Было найдено такое решение, которое сначала показалось каким-то странным, но оно хорошо работает. Мы берем и все, что относится к текущему контенту вкладки, изымаем из DOM'a. Мы получаем, например, по id этот элемент (какой-то контейнер), сохраняем в какую-то переменную, в некую очередь и выкидываем из DOM'a. Вместо этого вставляем какой-то новый элемент. Ссылка на старый у нас хранится. Она, естественно, не вычищается из памяти, потому что мы в любой момент можем ее использовать. Более того, это достаточно быстро работает. Мы можем достаточно быстро вставить его в DOM (то, что мы уже взяли), и оно вставится в том виде, в котором оно было.

Еще один момент — это переменные. Что делать с переменными. Большинство людей, когда пишут JavaScript, какие-то классы, функции и так далее, используют глобальные переменные либо закрывают их в какой-то scope. Но у них есть какие-то просто переменные, которые очень сложно выцепить. Мы решили, что все переменные, которые относятся к какой-то странице, мы будем называть «cur.что-то». У нас такой объект cur, в котором мы храним все подэлементы (все переменные, с которыми мы что-то делаем на этой странице). Как только пользователь переходит на другую страницу, мы все эти переменные собираем, складываем куда-то в очередь и потом, когда он нажал кнопку «назад», мы их оттуда достаем, складывая те, где он был, чтобы он мог вернуться вперед.

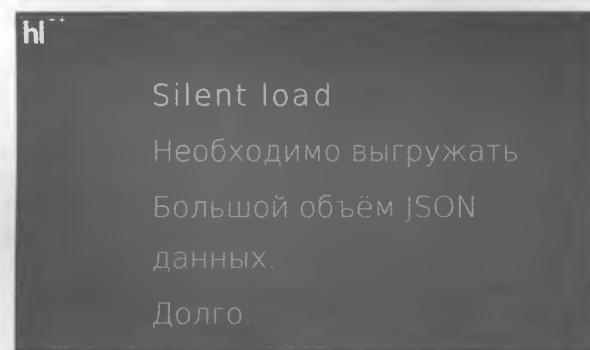
Таким образом, мы сохраняем полностью переменные, сохраняем DOM. Единственный момент — это всякие события, которые хотелось бы сохранить и которые навешаны не на вну-

тренние элементы, а которые снаружи. Очень часто на window нужно что-то повесить. Поэтому, естественно, нужны колбеки, нужно что-то обрабатывать.



Еще один хороший момент — это поиск в разных разделах. Например, первым это было сделано в разделе «Друзья». Была задача сделать так, чтобы поиск по «моим друзьям» был очень быстрым. Стало понятно, что на серверной стороне мы не сможем этого сделать. Мы не можем сделать достаточно быстрый поиск на серверной стороне так, чтобы пользователь пользовался этим, как фильтром. Чтобы он зашел в раздел «Мои друзья», начал что-то набирать и тут же увидел список. Чтобы он не чувствовал какой-то задержки.

Нужно выгружать всех друзей на клиента. Но это достаточно проблематично, когда друзей много (например, когда их 10 тысяч). Как это делается. Мы рисуем какую-то страницу. Естественно, мы должны в JSON'е выгрузить этих друзей. Чтобы искать на клиентской стороне по ним, мы должны проиндексировать их на клиенте, составить дерево. Понятно, что для этого нам нужно получить их в JSON'е. Но если мы получаем всех друзей в JSON'е, и их, например, 10 тысяч, то мы сталкиваемся с двумя проблемами.

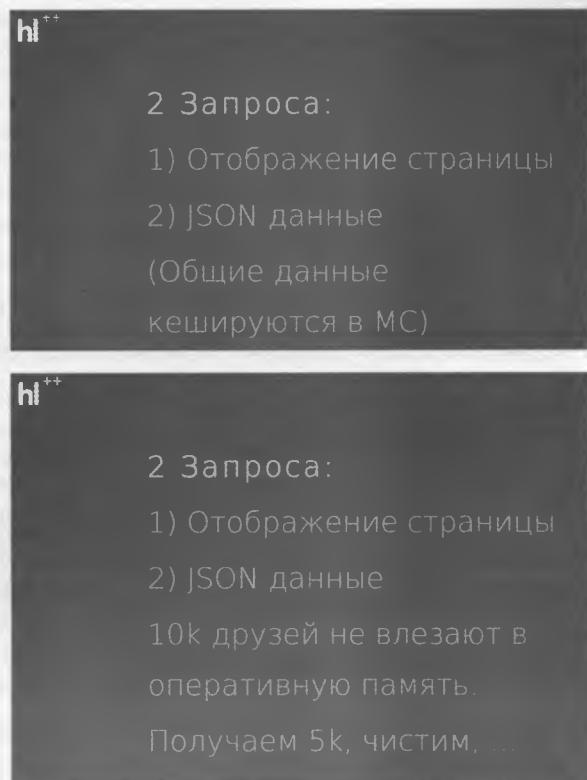


Первая проблема — это то, что этот JSON очень большой. 10 тысяч друзей — у некоторых моих ботов это было около двух мегабайт. Два мегабайта, которые нужно вылить на клиента при заходе в «Друзья», и только после этого показать ему страницу, это что-то убийственное. Мы можем показывать страницу по частям, но все равно это будет плохо работать.

Поэтому было решено делать несколько запросов. Мы делаем один запрос и показываем только страницу с п-ным количеством друзей (15 друзей). После этого мы делаем еще один запрос, чем слегка напрягаем Apache, зато мы кэшируем в memcached те данные, которые нам нужны посередине. То есть те данные, которые мы вытащили уже при первом запросе и используем при втором, мы кладем в memcached. Мы не достаем их второй раз, по крайней мере, из труднодоступных мест, только из memcached.

После чего мы получаем этот JSON, формируем его и отдаём его уже в фоновом режиме. Если пользователь куда-то перешел, если пользователь что-то нажал в этом интерфейсе, мы просто показываем загрузку до тех пор, пока не появятся данные, чтобы использовать быстрый фильтр.

Таким образом, удалось сделать достаточно быстрый интерфейс, который незаметно подгружает достаточно много данных. Если друзей мало, то это не используется, и все друзья получаются одним разом.



Это последний слайд. Наверное, надо рассказать что-то еще, так как времени много.

Есть ли какие-нибудь вопросы? Я бы оттолкнулся от какого-нибудь вопроса и продолжил рассказ.

Вопросы и Ответы

Вопрос из зала: По поводу long polling сервера. Что они вообще делают. Это прокси к чему-то или что? Например, откуда они берут данные?

Олег Илларионов: Да, я, кстати, не рассказал достаточно подробно об этом. Как это происходит.

Например, вы отправили кому-то сообщение. В момент отправки ваших сообщений мы формируем те данные, которые получит пользователь, которому вы отправили. В этот момент мы смотрим, какой у него long pack, получаем его long pack, берем шаблоны, составляем все, что нужно и отправляем в long poll сервер. Причем в независимости от того, есть ли этот пользователь или нет. Мы все равно отправляем в QE, чтобы потом это использовать.

Вопрос из зала: В смысле, это все клиент создает?

Олег Илларионов: Нет-нет-нет. Это, естественно, на сервере.

Вопрос из зала: То есть запрос сначала идет не на long polling, а отдельно.

Олег Илларионов: Нет-нет. Сами события в long poll сервер попадают из PHP. Мы в PHP получаем какое-то событие, формируем данные и отправляем их.

Вопрос из зала: Скорее всего, так, но просто на всякий случай, поговорить. На самом деле, речь идет о чем. Вы отправили данные на PHP, грубо говоря. Дальше в соответствии с какой-то логикой вы выплюнули все это в соответствующую очередь, которая потом ушла через long poll соединение клиенту. Правильно?

Олег Илларионов: Да.

Вопрос из зала: Есть ли у вас ситуации, когда нужно подсоединять не через PHP, а через какой-то отдельно стоящий сервис подключать. Не знаю, мультичат какой-нибудь. Или все идет только через PHP-frontend?

Олег Илларионов: Не совсем. На самом деле, есть два типа QE-серверов (серверов, которые управляют long poll'ом). Один из них является частью движка сообщений и отправляет события в изначально заданном формате сообщений, и это касается только сообщений.

Второй движок занимается этими же сообщениями для notify'ев. Но при этом он формирует и отправляет все остальные события, так как событий на сайте, на самом деле, очень много. Кто-то что-то залайкал, кто-то добавил что-то в newsfeed. Все это обновляется в реальном времени, поэтому требует событий, которые нужно обрабатывать на клиентской стороне.

Вопрос из зала: Получается, как-то симулируется ответ сервера в некоторые моменты. То же сообщение, например. При отправке ведь мы же должны показать список сообщений, а не ждать, пока это уйдет на сервер, придет в очередь, очередь рассосется long poll'ом и вернется обратно.

Олег Илларионов: Мы отдаем уже готовую информацию, но не какой-то раздел, а только то, что нужно поменять на странице. Для каждого случая это решается отдельно.

Вопрос из зала: Нет, я про то, что когда мы отправили сообщение, у нас должен появиться список сообщений, но он теоретически может долго идти до нас обратно. Есть какая-то эмуляция ответов?

Олег Илларионов: В плане по порядку?

Вопрос из зала: Просто пока придет очередь, что происходит? Мы же должны что-то показать. Мы показываем список сообщений уже с новым отправленным сообщением. А, там входящие появляются...

Олег Илларионов: Нет, естественно, мы берем и...

Вопрос из зала: Тогда понятно.

Олег Илларионов: Мы просто на лету формируем верстку относительно того, что нам нужно.

Вопрос из зала: Можно поподробнее про использование iframe'ов для подгрузки вместо Ajax'а? Очень быстро было.

Олег Илларионов: Хорошо. Сейчас расскажу. Идея в том, что мы не можем в достаточно большом количестве браузеров получать информацию, которая приходит по Ajax'у в течение

ее загрузки. Мы получаем этот контент только после того, как получили все, после того как сервер все отдал и прервал соединение. Это очень плохо.

Если у пользователя медленный коннект, на самом деле, нужен не такой уж и медленный коннект. Просто нужно, чтобы что-то было немного забито в сети. Это будет достаточно долго. Это будет заметное глазу время. Поэтому это нужно параллелизировать на части. Нужно отдавать контент частями. В противном случае пользователь будет замечать, что что-то работает не очень быстро.

Вопрос из зала: Это вы при первой загрузке страницы загружаете части, потом перемещаете в DOM? Или какая-то подгрузка идет. Сначала создается DOM-элемент, где iframe. Так?

Олег Илларионов: Когда просто страница открывается (не Ajax-переход, а просто она открывается), она открывается просто, как она должна быть. Когда пользователь сделал какой-то переход, перешел на какую-то ссылку, мы не перезагружаем страницу, мы остаемся на той странице, мы просто подгружаем элементы.

В этот момент мы, вместо того чтобы делать Ajax-запрос, делаем скрытый iframe, который имеет все параметры, аналогичные тому запросу, который бы мы сделали, если бы использовали Ajax для этого. Данные, которые приходят, выглядят примерно следующим образом: «top.какая-то функция callback» и набор данных, которые нужно использовать для отображения в каком-то блоке.

Вопрос из зала: Спасибо.

Олег Илларионов: Кстати, еще хотелось бы добавить такой момент. В вебе, на самом деле, рендеринг браузера не такой быстрый, как кажется. Мы привыкли считать, что браузеры отрисовывают все очень быстро, и это время можно не считать. На самом деле это не так. Они отрисовывают все за заметное глазу время. В принципе, это очень сильно ускоряется визуально, если мы рисуем по частям.

Если ответ у нас пришел не полностью, мы можем нарисовать верхнюю часть. Когда придет нижняя часть, мы гораздо быстрее отрисуем нижнюю часть, чем если бы рисовали и верхнюю, и нижнюю. Скорость рендеринга очень зависит от количества блоков, от того, что там внутри.

Вопрос из зала: Почему для этого надо использовать iframe, почему не использовать 3 разных Ajax-запроса?

Олег Илларионов: Потому что для этого придется очень мучить фронты и backend'ы.

Вопрос из зала: 3 iframe'а – точно так же будет 3 запроса.

Олег Илларионов: Нет-нет-нет. Идея в том, что когда мы загружаем iframe, он загружается браузером по частям. JS, который там есть, отрабатывает, не только когда он весь загружен. Если это разные блок и скрипт, они отрабатывают по мере загрузки. Закончился тег скрипта – он выкидывается, несмотря на то что последующие еще недогрузились.

Вопрос из зала: Сейчас производительность JavaScript'a постоянно улучшается. Когда вы планируете перейти на чисто клиентский рендеринг шаблонов и вообще всего?

Олег Илларионов: Мне кажется, это будет очень плавный переход. Точно так же, как с Ajax Layout'ом. Это был очень плавный момент. Не было такого, что раз, вчера сайт был обычный, а сегодня он весь на Ajax'е. Он переписывался по разделам. Специально для этого была придумана система, которая позволяла не конфликтовать обычному подходу и Ajax'овому. Переходы между разделами были Ajax'овыми, но как только пользователь выходил за этот круг, как только он выходил на какую-то старую страницу, она происходила обычным образом. Для этого мы просто руками на ссылки вешали callback'и. Так как мы взяли такое правило: все но-

вые переписанные разделы мы использовали без .php, делали rewrite. Чуть позже мы заменили этот callback на простую проверку. Если у нас .php, мы делаем обычный запрос, если у нас без .php (какая-то внутренняя страница, но без расширения), значит, мы делаем Ajax-запрос.

Вопрос из зала: Здравствуйте. Спасибо за хороший доклад. У меня есть вопрос немного нетехнический. Вы при реализации используете немного нестандартные методы (iframe'ы и так далее, что специфично для разных браузеров). Каким образом вы все это тестируете, используете ли вы какие-то специализированные средства?

Олег Илларионов: Мы используем «виртуалки». Естественно, сложно поставить на одну машину все браузеры. Мне не удавалось.

Вопрос из зала: Это понятно. То есть у вас используется ручное тестирование?

Олег Илларионов: Да. Довольно проблематично, например, в 6-м IE автоматизировать тестирование верстки. Я не знаю таких примеров. Хотя, наверное, при должном желании это можно сделать.

Еще хотелось бы рассказать такую вещь. Это решение одной проблемы, с которой все сталкиваются, и мало кто знает, как ее решить. Когда у нас нет History API, как сделать так, чтобы не было окна перехода. Пользователь открывает ссылку без Ajax'a. Хеш мы можем получить только на клиентской стороне, поэтому большинство загружает некую страницу, в которой он обрабатывает этот хеш, и после этого загружает в нормальную страницу.

Это выражается таким белым всплеском. Загружается, как правило, белая страница либо с какой-то заглушкой, после чего появляется нормальная страница. Для нас это было совершенно неприемлемо, потому что в тот момент, когда существовали и старые, и новые разделы, сложно представить, как бы отреагировали пользователи, если бы они переходили из одного раздела в другой, и посередине у них была бы какая-то заглушка.

Как мы решили эту задачу. Оказалось, что есть очень простой способ, который работает везде. Если мы в хедере до боди делаем JavaScript код, который вызывает «location=что-нибудь», то есть который меняет location. В этот момент браузер ничего не отрисовывает и по поведению ведет себя примерно так же, как серверный редирект. Это очень хороший хак, который спас всю затею.

Вопрос из зала: У меня целая серия вопросов. Для начала я хотел бы уточнить. Работа кнопки «назад» — это часть в History API? Когда я кликаю, происходит перехват метода, и браузер просто не переходит. Вы как-то через History API сделали?

Олег Илларионов: Да. Да. В старых браузерах это тоже работает за счет iframe'ов. Где нет History API, есть iframe'ы, которые меняют свои rule'ы. Страница не менялась, но менялись iframe'ы, меняется контент iframe'ов. Браузер запоминает это и при нажатии на кнопку «назад» не перезагружает страницу, а перезагружает iframe внутри страницы. Это можно отловить.

Вопрос из зала: Во время загрузки страниц таким хитрым способом у вас как-то производится загрузка счетчиков. Это загрузка страницы считается как загрузка страницы? Баннеры обновляются...

Олег Илларионов: Баннеры, естественно, обновляются, счетчики — тоже. Счетчики обновляются двумя способами. При любом обычном переходе получаются счетчики, но не все левое меню, а только цифры. Минимум трафика.

Второй способ — это QE. На большинство важных счетчиков стоит очередь. Как только они изменяются, приходит событие, и они обновляются.

Вопрос из зала: Когда вы манипулируете с DOM'ом, переносите страницу в какое-то хранилище и загружаете на ее место новую. Все события, которые были навешаны на клики,

на всплывающие окна, все это потом нормально возвращается назад в браузерах? С этим проблем нет?

Олег Илларионов: Если событие было на какой-то внутренний элемент, то оно возвращается. За исключением того, если это событие как-то баблится и так далее. С этим могут быть проблемы. Но нет таких случаев, когда это вело себя совершенно неадекватно.

Вопрос из зала: Проблем со снятием listener'a и навешиванием его обратно для очистки памяти не возникает?

Олег Илларионов: Возникает, конечно. Но это, в основном, верхние события. То есть события, которые навешиваются на window, на BodyNote. Еще такой интересный момент с BodyNote. Дело в том, что мы не используем document.body, потому что он не везде используется. Есть браузеры (например, IE), где когда вы заходите и смотрите на страницу, на самом деле, вы видите dif, у которого overflow out, у которого scrollbar. Вы видите не страницу, у которой scrollbar, а dif, у которого scrollbar.

Это сделано для того, чтобы спасти слои, которые открываются, когда вы, например, смотрите фотографию, и у которых своя верхняя прокрутка. Это можно очень элегантно сделать в Chrome. Можно достаточно элегантно, но не так, как в Chrome, сделать в Firefox'е, но без каких-то извращений. Например, в IE этого не сделать, а в Opera мы вообще не нашли способа сделать это достаточно элегантно, чтобы не было некоторых неприятных случаев с этим (например, чтобы не скролилась внутренняя страница при прокрутке слоя до конца).

Вопрос из зала: Большое спасибо.

Вопрос из зала: Я хотел бы задать вопрос по использованию веб-сокета. Будет ли он где-то использоваться или, может быть, уже где-то используется? Ваше отношение.

Олег Илларионов: Отношение хорошее. Очень хотелось бы иметь рабочие веб-сокеты в браузерах, но, к сожалению, сейчас с этим не все хорошо. Вы в курсе, что, например, Firefox отказался. Будем надеяться, что они передумают. Мы, например, считаем эту угрозу не такой страшной. С этим можно жить.

На мой взгляд, сокеты — это не обреченная технология. Она может жить. Когда-нибудь мы, возможно, будем это использовать. Но сейчас единственный способ сделать это достаточно эффективно — это long poll. Если заглядывать в будущее, скорее всего, это event driven. Всякие события, наверное, даже обгонят по развитию сокеты, потому что сокеты как-то застряли и непонятно, что с ними будет.

Вопрос из зала: Я хотел бы уточнить. Веб-сокеты обратно ввели в Firefox 6 со всеми починенными проблемами безопасности. Также в Chrome их уже ввели или введут в 16-й версии. И с Opera'ой что-то непонятное.

Ведущий: Небольшое объявление. У нас пропал следующий докладчик. Доклада про NoJS не будет, но вместо этого можно продолжить общение с Олегом. Мне кажется, про архитектуру vkontakte будет очень много интересных вопросов. Можно будет хорошо побеседовать. Спасибо.

Олег Илларионов: Я могу рассказать про NoJS. (Смех в зале, аплодисменты). На самом деле, в последнее время мы не очень активно его используем, но используем. В последнее время просто не было возможности его применить, за исключением отправки e-mail'ов, push-уведомлений и всякие такие вещи, связанные с Twitter'ом и так далее.

Например, отправка e-mail'ов великолепно работает просто потому, что там нет ничего сложного пока. Пока мы не сделали e-mail @vk.com, пока у нас e-mail'ы только с хешками, то есть вы можете отправить прямо из личных сообщений e-mail. Многие не замечали, потому что

надо еще догадаться вместо имени человека начать набирать e-mail. Но такая возможность есть. Ответ от этого пользователя будет получен и придет вам в личное сообщение (правда, без attach'ей).

Мы, на самом деле, уже давно мечтаем сделать @vk.com. Но системные администраторы не могут сделать это быстро, потому что там есть некоторые вещи с портами и так далее. Не могу сказать, когда появится эта функция. Наверное, довольно скоро. Мы, по крайней мере, довольно давно движемся к тому, чтобы организовать на уровне data-центров все необходимое, чтобы можно было это реализовать.

Мне кажется, самый лучший пример того, как можно использовать NoJS в таких проектах – это push-нотификации. Все знают, что iPhone поддерживает push-нотификации. Мы можем отправлять события на iPhone. Я могу рассказать, как это работает, потому что не все, наверное, знают. У Apple есть свой бинарный протокол. Это не так, как в Android'e, не нужно дергать какие-то скрипты. Вы подключаетесь к TCP-потоку и отправляете туда события в бинарном виде внутри бинарника. Правда, вы вкладываете еще JSON. Такая веселая смесь, но это очень хорошо. Мне это очень нравится.

Базовые вещи вы отдаете в бинарном виде, все остальное – в JSON'e. Очень удобно, достаточно экономично и за счет того, что это всего лишь один коннект, это работает достаточно быстро. Очень долго этим занимался один сервер. Потом просто из соображений того, что он может вдруг отвалиться, добавили еще, но это совершенно не требует каких-то больших нагрузок, несмотря на то что пользователей iPhone'ов великое множество, и всем нужно отправлять нотификации.

У нас были проблемы с доходом этих нотификаций. Они не всегда приходили. Сейчас эти проблемы были решены. Насколько я знаю, сейчас у всех работают push-нотификации на iPhone.

Вопрос из зала: Не у всех.

Олег Илларионов: Не у всех? Они вообще не приходят или что? Вообще не приходят. Странно. Мне тоже не приходят, но у меня девелоперская сборка, у меня они не должны приходить. Сколько я ни тестировал, ни проверял... Вы можете оставить мне свой id'ишник, я посмотрю в базе. Если они вообще не приходят, скорее всего, проблема на уровне того, что у вас в базе прописано, что вам не нужно отправлять push-нотификации. У нас были некоторые такие моменты, когда не совсем то писалось в базу. Были пользователи, которые попали в такое несчастливое число. Но я был уверен, что я всех этих пользователей проапдейтил.

Вопрос из зала: Раньше приходило, теперь перестало.

Олег Илларионов: Это очень печально. Финальный момент. Пришлось все переписать и заставить это работать хорошо. Я немного сменил протокол, и они стали сбрасывать коннект. Собственно, из-за этого были большие проблемы с доходом push-нотификаций. Они стали сбрасывать коннект, если с одного IP'шника я коннектчусь с двумя инстансами, с двумя разными сертификатами. Я делал это, чтобы подсыпал себе еще push-нотификации, потому что у меня девелоперская сборка, у которой другой сертификат. Они просто стали сбрасывать. Сервер все время реконнектился, отправлял несколько сообщений. Потом кому-нибудь из разработчиков отправлял эти сообщения, и все вырубалось. Это было очень печально, но сейчас такого не должно быть, я отключил всем разработчикам push-нотификации. Теперь все хорошо.

Вопрос из зала: Он отрубал вообще весь вконтакт, push'и от него полностью?

Олег Илларионов: Он дисконнектил. Apple очень весело себя ведет. Если что, он дисконектит. Причем в feedback ничего не приходит. У них два TCP-канала – feedback и основной stream. Но, в принципе, достаточно легко догадаться, почему он отключает. По крайней мере, если все делать правильно, они этого делать не будут. Но так как все работало, и изменен-

ние было с их стороны, пришлось немного подебажить и позэкспериментировать, чтобы дойти до этого.

Вопрос из зала: При отправке сообщения на e-mail, куда оно уходит? Можно указать полный адрес с @ или что?

Олег Илларионов: Идея в том, что когда вы набираете e-mail и отправляете сообщение, ему приходит e-mail со специального адреса. В адресе есть хеш. Мы проверяем, чтобы отсекать спам и все подобное. Нет спама и нет нагрузки.

Вопрос из зала: Я слышал, что у вас используется своя СУБД. Это правда?

Олег Илларионов: Это правда.

Вопрос из зала: Вы можете рассказать, что лучше и какие сравнительные характеристики с чем-нибудь, хотя бы с MySQL?

Олег Илларионов: Давайте я расскажу, это очень интересно. У нас используется и MySQL для каких-то задач. Изначально MySQL использовался для всего. Но в определенный момент стало понятно, что MySQL справляется не со всем. Первая вещь, которая была переписана, это личные сообщения. С ними было больше всего проблем.

Основная причина — поиск. Хотелось искать по сообщениям. В MySQL начать искать по сообщениям каждого пользователя оказалось нереальной задачей. Был написан специальный движок. Он называется *text engine*. Он написан специально для личных сообщений. В нем много чего не предусмотрено. Редактирование было не предусмотрено до последнего момента. Но так как *text engine* — это не только личные сообщения, но и стены, то недавно появилось редактирование на стенах. Это было очень серьезное изменение в движке.

Это специальный движок, который позволяет получать выборки по определенному ранжу очень быстро и имеет такую структуру, которая оптимизирована для этих вещей. Когда есть какой-то хозяин (owner) и есть какие-то его записи.

Вопрос из зала: Не могли бы вы рассказать, как у вас технически устроена отправка почты юзерам. В момент ее генерации, как почта путешествует внутри ваших площадок, на которых стоят сервера. Это не вопрос, как попасть в inbox или в спам. Как вообще вы все это мониторите и понимаете, что у вас все хорошо.

Олег Илларионов: Отправка, естественно, не очень прямая. Мы не можем просто взять и отправить с того же сервера почту по понятным причинам. Есть сервера, которые выполняют разные задачи. Если один сервер будет выполнять все задачи, то сисадмины повесятся.

Когда отправляется почта, просто кладется запись в очередь. Для очереди, кстати, сделан свой движок, который оптимизирован под очереди. Дело в том, что все отправляемые письма почему-то хочется хранить, и поэтому MySQL тоже плохо справлялся с тем, чтобы хранить все.

Вопрос из зала: Вы храните прямо исходник письма, который вы сгенерили, с html, заголовками — все вместе?

Олег Илларионов: Да. Насколько я понимаю, это было по той причине, что периодически отваливалась отправка почты когда-то давно и хотелось потом отправить все, что не удалось отправить. Это кладется в специальный движок, который это хранит до того момента, пока другой сервер не обратится за этим. Но это, как правило, происходит очень быстро. Он обращается за этим письмом, чтобы отправить его. Это уже специальный сервер, настроенный исключительно для отправки почты.

Вопрос из зала: Правильно ли я понял, что у вас есть некоторые машины, которые, видимо, сами генерят html и txt-версии писем с нужными заголовками. После этого кладут их в систему

му, которая база/небаза, в ней лежат полные исходники писем. Есть совершенно отдельные выделенные машины, которые из той очереди забирают эти сгенеренные и по факту делают только команду mail, отправляя в ваши NTA.

Олег Илларионов: Нет, все с другой стороны. При отправке сообщения все необходимые данные кладутся в эту очередь. Потом сервер, который отправляет, делает команду mail, составляет шаблоны, конфигурирует из тех данных, которые он вытащил, и отправляет. Правда, это касается только уведомлений.

Та почта, которая отправляется на какой-то определенный e-mail из личных сообщений, на самом деле, работает совершенно по-другому. Ее отправка написана на JS. Это отдельный сервер в другом месте с другими IP'шниками, который через наше API получает эти письма и отправляет их. Это совершенно внешняя система. Она сделана для того, чтобы в случае если все отправляемые письма будут характеризоваться как спам (чего мы очень боимся), это не стало критичным для отправки нотификаций. Чтобы были совсем разные IP'шники и все было совсем по-разному.

Вопрос из зала: У вас куча всяких IP'шек из разных подсетей...

Олег Илларионов: Нет, их всего несколько. Есть такая маленькая подсеть, а есть все остальное.

Вопрос из зала: Сколько машин, если не секрет, занимается отправкой почты, на которых NTA крутится?

Олег Илларионов: Отправкой нотификаций занимается достаточно большое количество машин, но я совершенно не в курсе сколько.

Вопрос из зала: Это десятки, сотни?

Олег Илларионов: Это десятки, но не сотни. Отправкой e-mail'ов занимается один сервер. Надо его продублировать, а то вдруг он упадет.

Вопрос из зала: Какие объемы писем, если не секрет, отправляете каждый день? Просто я читал доклад про 100 миллионов писем каждый день. Мне интересно, если мы с вами по меряемся, кто окажется круче на цифрах. (Смех в зале).

Олег Илларионов: Невероятное количество. Я заикаюсь, когда такие цифры вижу. Очень много нотификаций за счет того, что очень долгое время регистрация на сайте была достаточно долгое время через e-mail, и всем этим людям надо отправлять нотификации. Это дикое количество.

Вопрос из зала: То есть у вас, извиняюсь, «длиннее», я правильно понял? (Смех в зале, аплодисменты).

Олег Илларионов: Я, если честно, не знаю этого числа, потому что я не запомнил количество нулей. Я его как-то видел.

Вопрос из зала: Просто чтобы добить вопрос про почту. Каким образом вы мониторите? Есть какой-нибудь отдельный человек или группа, которые занимаются вопросами, как эти письма правильно отправить, доставить, как их правильно генерить?

Олег Илларионов: Нет, конечно. У нас маленькая команда. Есть просто статистика по количеству отправляемых писем. Начальство периодически смотрит на статистику. Если там что-то не так, то все плохо.

Вопрос из зала: Что можно увидеть из статистики, которая представляет из себя прямую линию, на ней нотификации о новых сообщениях — 5 миллионов. Вот она прямая последние два месяца. Что там можно увидеть?

Олег Илларионов: Она не прямая. Сайт постоянно растет достаточно большими темпами, и это число увеличивается.

Вопрос из зала: То есть я правильно понимаю, что единственный пока способ мониторинга, что вообще все хорошо. Скорее, вы меряете генерацию, получается, писем.

Олег Илларионов: Идея в том, что если что-то случилось с сервером, то придет sms-ка сисадминам. Это, конечно, не самый лучший способ решить проблему, потому что может произойти что-то не совсем с сервером. Но пока это достаточно хорошо работает. Как правило, если не отправляются письма, то сисадмин уже в курсе.

Вопрос из зала: Понятно. Но вы хотя бы меряете эту очередь? Грубо говоря, у вас стоит очередь заявок на генерацию писем про новые комменты, про еще что-то новое. Вы меряете, какой у нее объем? Вдруг там стоит 100 миллионов, вы их накопили за какое-то время.

Олег Илларионов: Да, я точно не знаю, стоит ли в этом случае очередь. Но у нас есть такой специальный скрипт `watch doc`, который проверяет разные числа, смотрит, чтобы они были достаточно высокими, и шлет sms-ки, если что.

Вопрос из зала: Хорошо. Вы вроде очень подробно рассказали. Спасибо.

Вопрос из зала: Расскажите, пожалуйста, какие сервера очередей используются и в каких случаях?

Олег Илларионов: Вы имеете в виду — для внутренней передачи данных?

Вопрос из зала: Да.

Олег Илларионов: Мы не используем какой-то Open Source'ный формат, как делают многие. Мы решаем задачи такими, не самыми оптимальными способами, зато самыми быстрыми, потому что, как правило, это вещи, которые не касаются высоких нагрузок. За исключением e-mail'ов, где свой движок. Там понятно — там много писем, и это, действительно, высоконагруженная штука.

Во всех остальных случаях это всякие интеграции, парсинг RSS, еще что-то. Это, как правило, обычный MySQL. Даже импорт в Twitter — это обычный MySQL. Там специальная табличка, которая заточена на конкретные задачи. Она выступает в роли очереди.

Вопрос из зала: Давно хотел спросить. Зачем вы ломаете в History API логику поведения браузера? Если в ленте новостей открыть фотографию, она открывается в LightBox'e, потом ее закрываем и нажимаем «назад», мы попадаем не на предыдущую страницу, а в LightBox.

Олег Илларионов: Да. Действительно, ломаем. Мы считаем, что так лучше.

Вопрос из зала: То есть вы считаете, что ломать стандартную логику браузера лучше? Вы знаете лучше.

Олег Илларионов: Да, мы так считаем (смеется). Это такой спорный момент. Не все с этим согласны, естественно. Я считаю, что это момент, который нужно немного дорабатывать, нужно находить золотую середину. С одной стороны, не очень хорошо, когда пользователь просмотрел целый альбом, а потом, например, хочет вернуться на человека. Нажимает «назад», егодвигает назад по всему альбому. С другой стороны, может быть, пользователь хотел как раз назад перейти.

Это разные психологии, разные вещи. Был очень забавный момент, мы периодически шутили. В «Facebook» был забавный баг: если из «ВКонтакте» перейти на «Facebook», открыть там фотографию, что-то покликать и нажать «назад», то пользователь попадал на «ВКонтакте».

(Смех в зале).

Ведущий: У нас появился человек, который расскажет про Node.JS. Олег, огромное спасибо. Ты нас очень выручил. По-моему, всем было очень приятно пообщаться. Ждем снова.

Олег Илларионов: Всем большое спасибо. Рад был всех видеть.

Интеграция открытых технологий и взаимодействие со сторонними проектами в условиях высоких нагрузок

Олег Илларионов



HighLoad++

Интеграция открытых технологий и взаимодействие со сторонними проектами в условиях высоких нагрузок.

ВКонтакте

Олег Илларионов: Начать хотелось бы с беседы.



HighLoad++

XMPP сервер:

- реализация протокола
- интеграция с ВКонтакте

ВКонтакте

Дело в том, что задача создать XMPP сервер стала для нас нестандартной. В целом все довольно привычно и просто. XML — мы привыкли так общаться с пользователем — все довольно стандартно, за исключением того, что в данном случае мы вынуждены держать соединения.

Каждый пользователь у нас висит. Он может висеть час, два, день, неделю. Нам нужно держать много коннектов. PHP не смог помочь нам в этом. Пришлось искать другие технологии.

hl⁺⁺ **HighLoad++**

XMPP сервер:

- реализация протокола
(только основной протокол)
- интеграция с ВКонтакте
(необходимо полностью интегрировать контакт лист, систему сообщений и предусмотреть кеширование)

ВКонтакте

Сначала мы стали рассматривать готовые сервера, которые уже были: EJabber, Jabber E2 и так далее. Но смотря на все это и оценивая ситуацию, мы пришли к выводу, что реализовать XMPP протокол, работать с ним не так сложно, как интегрировать все с существующей инфраструктурой: с сервером личных сообщений, с контакт-листом. Нужно оповещать, когда кто-то вошел в сеть, кто-то вышел из сети. Нужно делать многие вещи.

Поэтому мы поняли, что гораздо дешевле и проще будет соорудить что-то свое. Мы стали думать, что нам нужно.

hl⁺⁺ **HighLoad++**

Нужна была платформа:

- Язык высокого уровня
- Высокая скорость
- Неблокирующий I/O
- Наличие инфраструктуры

ВКонтакте

Нам нужна была платформа — некий язык, который бы выручил нас в этом. На создание сервера было отведено очень мало времени — всего месяц. Не было особо времени на размышления, и мы стали смотреть на инструменты для быстрого программирования, вещи, на которых можно написать что-то быстрое, и оно будет работать хорошо.

Таким инструментам мы предъявили следующие требования.

- Это должен быть язык высокого уровня (Python, JavaScript, PHP — что-то из этой категории).
- Он должен довольно быстро работать, не сжирать все CPU, чтобы нам хватило нескольких серверов для Jabber сервера.
- Так как у нас очень много коннектов, пользователей, мы не можем работать с базой, с диском, с блокировками. Нам нужно использовать неблокирующий ввод-вывод.

- Последнее, что нам было нужно, это наличие инфраструктуры. У нас используется много баз (MySQL, Memcached). Нам нужна инфраструктура, чтобы мы могли работать на этой платформе с тем, что у нас было.



hl⁺⁺ HighLoad⁺⁺

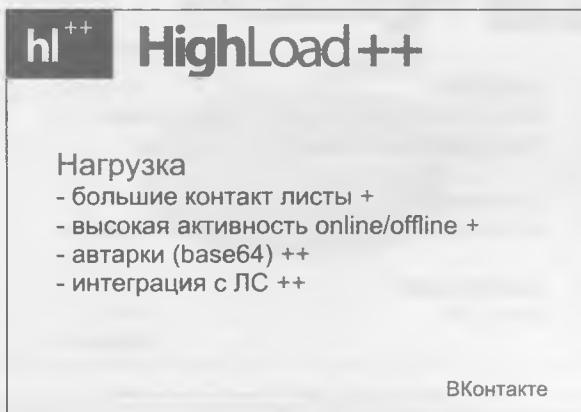
node.js

- + Язык высокого уровня
- + Высокая скорость
- + Неблокирующий I/O
- + Наличие инфраструктуры

[ВКонтакте](#)

Решение этого стало node.js. Это быстрая платформа. На ней можно создавать веб-сервера очень быстро. Фактически Jabber сервис был написан всего за 2 недели. Остальное время ушло на дебаг, на ускорение, на тестирование и на прочие вещи. Он не блокирует и обладает очень полной инфраструктурой. На данный момент все, что нам было нужно, было.

Даже MySQL, работа с которой славится не очень хорошими тонами, там была и работала. Все было стабильно и хорошо.



hl⁺⁺ HighLoad⁺⁺

Нагрузка

- большие контакт листы +
- высокая активность online/offline +
- автарки (base64) ++
- интеграция с ЛС ++

[ВКонтакте](#)

Дальше я хотел бы рассказать, с какими проблемами мы столкнулись. Самое страшное — это нагрузка. Когда мы тестировали все это на локальном компьютере, все было хорошо. Все очень быстро, все написали, все работает. Начали тестировать. Банальный скрипт, который коннектится, что-то делает, получает (неразборчиво) — сразу выявились проблемы.

У нас нетипичная ситуация, мы не обычный XMPP сервер. У нас большие контактные листы. Я тестировал на большом количестве соединений от самого себя и видел, что это не очень хорошо работает, потому что контакт-лист — за 400 друзей.

Высокая активность online/offline. Это тоже очень большой минус, потому что люди все время входят и выходят. В основном, это обусловлено тем, что сессия на сайте длится всего

15 минут. Если человек отошел от компьютера на 15 минут, потом вернулся и ткнул еще на какую-то кнопку, он вышел-вошел. Эти выходы-входы постоянно повторяются. Получается, что несколько раз за минуту кто-то выходит и кто-то входит. У нас постоянный обмен трафиком с клиентами.

Следующий момент — это аватарки. С ними очень непросто. Мы не можем взять и сконвертировать для каждого пользователя отдельную аватарку. XMPP — это редкий сервис, его не все используют. Если мы будем конвертить все аватарки, это будет слишком тяжело, тем более что клиенты (большинство из них) кэшируют аватарки у себя.

Все, что нам было нужно, это ужимать их налету. Причем отправлять их клиенту мы можем только в формате base64, потому что есть некоторые возможности. Протокол позволяет посыпать просто ссылки на них, но большинство клиентов в таком виде не воспринимает аватарки. Если мы хотим, чтобы это работало везде, нам нужно их в XML base64. Все это надо делать налету.

Человек просит свой контакт-лист, и для каждого пользователя мы ему — base64, его аватарку, все 400 человек. Все это должно произойти за несколько секунд. Таких пользователей может быть много.

Последний пункт — это личные сообщения. Архитектурно нельзя было взять и один коннект на тысячу человек для личных сообщений. Настолько сложна архитектура на данный момент, что для каждого пользователя нужно держать свой коннект для личных сообщений. В целом это довольно проблематично. Когда мы имеем 10 тысяч пользователей на одном сервере, нам тяжело держать 10 тысяч соединений с AIM'ом серверами. Их меньше, тем не менее, для каждого пользователя должно быть свое соединение.

The screenshot shows the HighLoad++ monitoring interface. At the top left is the logo 'hl++'. To its right is the title 'HighLoad++'. Below the title, there's a section titled 'Цифры:' (Digits:) containing text about user counts and server setup. To the right of this text is a link 'ВКонтакте' (Vkontakte).

Параметр	Значение
60000 ~ 80000	пользователей online
пик - 150000	пользователей online
(5 серверов)	
по 4 воркера node.js	на каждом сервере
3 БД (mysql)	

Сразу хотел бы поделиться цифрами, чтобы потом их обсудить. Что мы имеем в итоге.

В среднем 60-80 тысяч online. Пик был после написания в блоге, на следующий день вечером — 150 тысяч. Это было тяжело. На тот момент работало 4 сервера на полную катушку. Все было нормально, но я уже побаивался. Я смотрел на телефон каждые 15 минут, проверял, все ли работает.

В целом, когда мы только писали об этом на Хабре, реакция была совсем маленькой. Все совершенно спокойно держал один сервер. Два других (на тот момент 3 сервера у нас было) мы использовали для выката апдейтов. Очень просто: спереди был (неразборчиво) прокси, мы что-то изменили — на другой сервер, и всех пускаем на другой сервер. Старые оставили на старом. Очень удобно, потому что таким образом мы совершенно безболезненно выкатываем апдейты.

Дело в том, что на node не так просто апдейтить код. Можно, конечно, делать hot reloading, перезаливать все заново, переопределять все функции. Но это не совсем правильно. Тогда дебажить все это было бы совершенно невероятно, потому что утечки памяти были бы гарантированы.

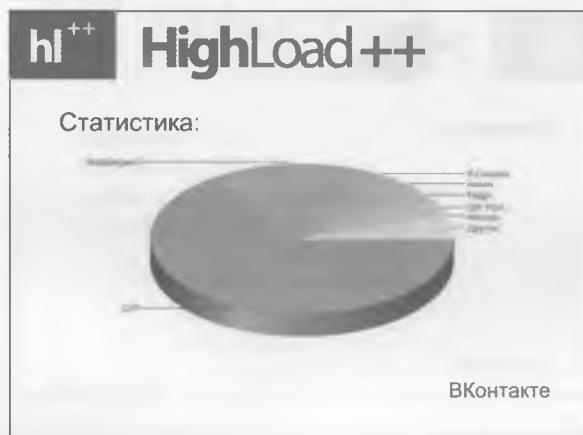
Базы данных используются, в основном, для кэширования и для хранения групп. У каждого пользователя есть свои группы. Каждый пользователь может как-то разбить свой (неразборчиво) — кого-то в одну группу, кого-то в другую.

Сначала мы использовали стандартные списки ВКонтакте. ВКонтакте можно создать список, люди будут в разных категориях: этих — сюда, этих — сюда. Но мы пришли к выводу, что это неправильно. Списки ВКонтакте — это такая личная вещь. Они используются для ограничения приватности. Эти же списки нужны для показа того, какие люди к каким относятся.

Поступило много жалоб. Люди не хотели видеть списки приватности у себя в клиенте, чтобы друзья заходили и видели, кто у них где. Это было нехорошо. Поэтому пришлось быстренько «на коленке» написать все это дело и хранить в MySQL.

Сначала думали использовать Mongo, но умные люди сказали, что не стоит две рискованные технологии использовать одновременно. Это опасно. В принципе, это было правильно, потому что MySQL работал хорошо. На этих же пяти серверах, 3 базы (они раздроблены). Там 3 сервера помощнее, 2 — послабее. На трех помощнее стоят базы.

Надо сказать, что в момент низкой нагрузки MySQL ест больше, чем node. Когда нагрузка большая, то node начинает есть больше, чем MySQL.



Сразу хотелось бы определиться со статистикой. Все было самописное, мы очень просто могли собирать статистику. Хотелось бы поделиться, с чего люди заходят. Если честно, я был удивлен. Хотя бы тем, что Adium обогнал мой любимый Pidgin. Но дело даже не в этом. Я даже не знал о существовании MobileAgent'a, например. Но в целом люди заходят через QIP. Это очень хорошо. QIP хорошо относится к нашим серверам, гораздо лучше, чем Miranda, с которой трафик ужасен.

С какими проблемами мы столкнулись. В node сложно с памятью. Она не течет, там хороший garbage collector, но нужно быть очень аккуратными, потому что возникает очень много corner case'ов. Мы пишем сложную систему — XML, разные случаи, разные клиенты. Все они по-разному работают, и ошибки неизбежны. В определенный момент возникает ошибка. У кого-то в этот момент оборвалось соединение, тут сработал callback, нужно ему что-то дать, там чего-то нет — и все обвалилось.

В этот момент нельзя предсказать, что из этого выйдет. Может случиться так, что память течет. Причем долгое время память утекала, и мы видели, что у нас остаются висящие пользователи.

Что мы сделали. Мы написали цикл, который пробегается по всем пользователям, смотрит, когда в последний раз от этого пользователя приходило сообщение. Что происходит. Если не слал какое-то время (по-моему, там стоит 180 секунд), то он вырубается. Периодически, раз в 50 секунд посыпаются пинги. Если он не отвечал на пинги (ничего не отправлял за эти 180 секунд), он полностью удаляется отовсюду. Таким образом, удалось избавиться от всех утечек. Все перестало течь.

Другая проблема — это OpenSSL. В целом node — готовый инструмент, его можно использовать на продакшне, когда речь не заходит про OpenSSL. С ним возникает куча проблем. Из-за этого приходилось ковыряться в коде node, в коде C++, надо было даже отправить патчи в сообщество. Но это не все.

По сути, мы имеем свой небольшой (неразборчиво), где исправлено много чего по работе с OpenSSL. Исправлено грубо, поэтому это все незарелизено. По сути, эту часть нужно очень сильно рефакторить в node. Но нам удалось добиться некоторой стабильности.

В целом, конечно, это ужасная ситуация, потому что очень многие клиенты используют OpenSSL по default'у (примерно 20%). OpenSSL довольно тяжело работает, он жрет процессор, жрет ресурсы и является причиной 90% багов.

Так как времени мало, хотелось бы сразу продолжить.



Дальше хотелось бы рассказать о виджетах и о кросслодеменном взаимодействии. Мы очень много двигаемся в этом направлении. Это одно из важных направлений, потому что в последнее время мы стремимся к открытости и хотим дать веб-мастерам больше возможностей.

Как это можно сделать. По сути, виджет — это довольно простая штука. Мы даем веб-мастеру кусок HTML, он размещает его у себя и получает что-то подобное тому, что изображено на слайде.

Но на деле все не так просто. Мы не можем дать веб-мастеру информацию о пользователе. Мы не можем дать ему возможность что-либо «заливать» без ведома пользователя, комментировать что-то без его ведома (особенно если это пойдет к нему в статус). Мы должны разграничить доступ. Как это делается. Естественно, это делается iframe'ом на другом домене. Это довольно просто.

Но делая это, мы лишаем доступа себя. Нам тоже нужно общаться с родительским окном, нам тоже нужно передавать информацию.

The screenshot shows a presentation slide with a dark header containing the text 'hl++' and 'HighLoad++'. Below the header, there is a list of 'Problems' with one item: '- изоляция данных'. In the bottom right corner of the slide area, the text 'ВКонтакте' is visible.

Например, нам нужно ресайзить iframe, еще что-то передавать разработчику. Что же делать. Нам нужно использовать какие-то инструменты, чтобы передать данные наверх — из iframe'a наверх. Что можно сделать. Есть стандартная функция post message, которая работает в большинстве браузеров и решает эту проблему. Но она работает в большинстве браузеров — не во всех.

The screenshot shows a presentation slide with a dark header containing the text 'hl++' and 'HighLoad++'. Below the header, there is a list of 'Problems' with three items: '- изоляция данных', '- кроссдоменное взаимодействие', and '- кросбраузерность'. In the bottom right corner of the slide area, the text 'ВКонтакте' is visible.

Кросбраузерность становится очень большим камнем преткновения в этом деле. Можно видеть, что здесь, в принципе, (неразборчиво) браузер — Firefox 3, Chrome, Opera, IE. Но пользователей более ста браузеров — около 10%. Это много. Поэтому нужно искать выходы в разных ситуациях.

hl⁺⁺ **HighLoad++**

- кроссдоменное взаимодействие
postMessage (FF >= 3, Chrome, Opera >= 9, IE >= 8)
- Как быть с остальными браузерами?

[ВКонтакте](#)

Сначала мы решили поступить очень просто и взяли готовую библиотеку. В самом начале был Flash-посредник. Мы его использовали там, где нет postMessage, но быстро поняли, что это решение неидеально, потому что Flash тоже есть не у всех, это работает долго, заставляет вкладки тормозить.

hl⁺⁺ **HighLoad++**

- flash посредник
работает везде где есть flash, но уступает по скорости.

[ВКонтакте](#)

Мы стали использовать готовые технологии. Мы взяли easyXDM. Это такая библиотека, которая разрешала все наши проблемы. Мы поставили ее — и все, казалось быть, заработало. В обычных приложениях ВКонтакте (iframe-приложениях) все работает до сих пор. Там стоит easyXDM. Хотя, на самом деле, это будет работать так всего лишь еще неделю.

Дело в том, что там есть клиентская библиотека XDM connection. Мы обновили версию. Она может работать с новой библиотекой, которую мы написали вместо easyXDM. Но так как у всех еще кэш, мы не можем взять и обновить. Мы ждем, пока кэш у всех очистится. Где-то две недели назад это был сделан — еще через две недели можно будет смело все обновлять. Кэш уже ни у кого не останется.

Был easyXDM, но были проблемы. Когда мы стали использовать его в виджетах, то нам начали жаловатьсяся мастера, что ничего не работает. Были разные ситуации. У кого-то переопределен JSON, причем переопределен в самый неожиданный момент (сначала JSON был нормальным, а потом — бац — он подключил какую-то библиотечку, там новый JSON, у которого почему-то нет функции `parse_ini_file()`). Также были проблемы: у него был JSON, он стал переопределенным, там даже есть функция `parse_ini_file()`, но работают они как-то странно. Например, списки преобразовываются в строчку.

Были странные проблемы, например, с (неразборчиво). Вроде как все нормально, но вылетает ошибка — ничего не работает. Дело в том, что разработчик там переопределил (неразборчиво), на который при кириллице выдает страшную вещь — и все вылетает.

Мы поняли, что так продолжаться не может. Мы не можем использовать эту библиотеку, потому что она вся основана на куче кодов, она сделана неправильно. Мы решили написать нечто свое.

hl⁺⁺ HighLoad++

- hash родительского окна
медленно работает, заметен пользователю, может конфликтовать с библиотеками пользователя, использующими hash

[ВКонтакте](#)

Мы стали смотреть, какие есть способы. Самый примитивный способ — это hash родительского окна. Как мы можем обмениваться информацией с iframe'ом. Мы можем менять hash родительского окна. Мы можем что-то записать туда, быстро это считать, пока не поздно, и быстро убрать. Если мы будем делать очень быстро, то клиентские скрипты будут работать. Они не заметят этого.

Тем не менее, это ужасный способ, потому что пользователь будет видеть, что у него там что-то бегает, что-то передвигается. Это отвратительно.

hl⁺⁺ HighLoad++

- hash дополнительного iframe
медленно работает, требует дополнительного файла на домене сайта.

[ВКонтакте](#)

Более продвинутый способ — использовать дополнительный iframe с hash'ом. Внутри нашего iframe мы создаем еще один iframe (уже звучит ужасно, но это не самое ужасное). В этом дополнительном iframe'е мы меняем hash и передаем ссылку родительскому. Родительский может дотянуться через iframe внутрь и увидеть hash, потому что они на одном домене.

Но у этого способа есть ужасный минус. Разработчику приходится вешать себе на host специальный файл. У нас он назывался easy rexEver. Собственно, Easy XDM тоже его использовал. Раз-

работчикам, чтобы их сайты работали в старых браузерах, требовалось положить к себе в файлы и виджеты, и open API. В принципе, многие до сих пор кладут, хотя из документации мы это убрали.

Это тоже не совсем правильно. Все должно быть проще.

The screenshot shows a section of the HighLoad++ documentation. At the top left is the logo 'hl ++' followed by the title 'HighLoad++'. Below the title, there is a list item '- nixTransport' with the note 'IE <= 7'. A detailed note follows: '(созданная через VB функция доступна у window.opener фрейма)'. At the bottom right of the screenshot area is the text 'ВКонтакте'.

Мы стали смотреть, как это можно сделать по-другому, какие еще способы есть. Выяснилось, что есть такая вещь, как nixTransport. Эта идея появилась в Easy XDM, начиная с какой-то версии. Оттуда мы ее и взяли.

Это такая особенность Internet Explorer'a, что если мы создаем на Visual Basic какую-то функцию, то мы можем до нее дотянуться. Мы создаем ее сверху, а изнутри, через window.opener мы можем до нее дотянуться, что-то туда передать, что-то выполнить. Это позволяет нам общаться. Причем быстро, потому что это нативная передача, без всяких посредников.

Что самое главное — это работает и в 6-м, и в 7-м IE. На 5-й мы не смотрим, потому что его практически не существует, слава Богу. Но это решает проблему для IE'шников. Мы стали использовать этот способ.

Вторая категория браузеров, которая нас волновала больше всего, это Firefox.

The screenshot shows a section of the HighLoad++ documentation. At the top left is the logo 'hl ++' followed by the title 'HighLoad++'. Below the title, there is a list item '- frameElement' with the note 'FF <= 2.0'. A detailed note follows: '(присвоенные к фрейму функции доступны из фрейма через window.frameElement)'. At the bottom right of the screenshot area is the text 'ВКонтакте'.

У Firefox'a тоже нашлась замечательная особенность — frameElement. Это очень интересное свойство. Насколько я знаю, в новых версиях его уже нет. В новых версиях есть postMessage, там все хорошо, мы можем передавать строчки. В старых мы можем присвоить iframe'у какую-нибудь функцию или несколько функций.

Из внутреннего iframe'а мы можем обратиться к этому iframe'у, к этим функциям через frameElement. Мы берем window.frameElement и эти функции — и они работают.

Конечно, для безопасности там есть небольшая защита. В window.name мы кладем специальный ключик и по нему уже обращаемся, чтобы кто угодно не мог вызывать родительскую функцию (какой-нибудь iframe на странице из какого-то таргетирга) и всех огорчить.

По сути, мы перекрыли почти все браузеры, потому что осталась только Opera (причем начиная с 9-й версии, она уже работает). С Opera'ой, конечно, отдельная история. В Opera 9.5 было очень много жалоб, потому что каким-то образом на ней сидит очень много людей. Это удивительно.

Она вылетала, когда postMessage использовался неправильно. Там есть postMessage, но не дай Бог как-то проверить, что он существует (там, где его нет), все вылетит напрочь. Это ужасно. Нам писали такие гневные письма: «Что за дела?», мы «убиваем браузер».

Но в Opera'e, в принципе, все работает, а 8-я Opera очень низко. Мы смотрели по статистике — ее практически не существует. Это какие-то очень редкие пользователи. Если они не увидят какой-то виджет, то, мне кажется, можно не обращать на них внимания. Например, в нашем частном случае гораздо больше пользователей на работе просто заблокировали Контакт и они тоже не видят виджетов.

Кстати говоря, с этой проблемой мы отчасти боролись. Дело в том, что мы не можем держать виджеты на другом домене, потому что нам нужно иметь куки. Будет странно, если всех пользователей мы будем авторизовывать, перебрасывать. ВКонтакте у всех есть куки, поэтому виджет он должен авторизованно.

Проблема была в том, что если ВКонтакте заблокирован, то браузер не мог потянуть библиотеку Open API. Сначала мы об этом не подумали. Мы вообще сначала не подумали о том, что Контакт может быть где-то заблокирован. (Смех в зале).

Но потом нам стали жаловатьсяся. Оказалось, что есть страница, сверху подключен скрипт — бац, он не грузится. Что происходит? Страница не грузится, потому что браузер не пойдет дальше, пока не догрузится этот скрипт или пока (неразборчиво).

Что мы сделали. Мы написали в документации, во-первых, один из способов — асинхронная загрузка (когда скрипт подключается не сразу, а по тайм-ауту). Разработчики пишут на сайте примерно 6 строчек кода, которые вызывают (неразборчиво). Это один из способов.

Второй способ. Мы открыли всю статику через user API. Тоже удобно, потому что user API пока никто не (неразборчиво), слава Богу. Userapi.com или userapi.ru — просто мы взяли другие домены. Это тоже решило проблему.

The screenshot shows a section of the HighLoad++ library documentation. At the top, there's a logo with 'hl++' and the text 'HighLoad++'. Below it, a bullet point says '- easyXDM'. Further down, there's a block of text: 'Использует postMessage, nixTransport, frameElement' followed by 'размер: 17 кб' and 'min: 4.5 кб'. At the bottom right of this block, the word 'ВКонтакте' is visible, indicating a download or usage example for that platform.

Теперь некоторые данные: что в итоге получилось. У нас есть easyXDM, он весит 17 килобайт, в минифицированном виде он весит 4,5. fastXDM — в минифицированном варианте он весит совсем не 4,5. Я просто забыл измерить.

hl⁺⁺ HighLoad⁺⁺

- fastXDM

Использует postMessage, nixTransport, frameElement
размер: 6.3 кб
min: 4.5 кб

не конфликтует при переопределённом JSON, не
использует onDocumentLoad

[ВКонтакте](#)

В неминифицированном виде он весит 6,3 килобайта. Но в минифицированном виде он будет весить совсем мало, потому что мы сделали очень хитрую вещью.

Мы решили, что у нас есть хороший и нехороший браузер. В хорошем браузере мы просто можем использовать небольшую обертку вокруг стандартного postMessage. В нехороших браузерах мы можем загружать дополнительную библиотеку, где будет все остальное, чтобы у нас было минимум кода.

Таким образом, если в браузере нет JSON или еще чего-то, он автоматически считается нехорошим, несмотря на то что он в целом хорош. Если JSON переопределен, подключена хитрая библиотечка, которая бы переопределила JSON, то браузер тоже считается нехорошим. Библиотека будет подгружаться в нормальный JSON (библиотека называется JSON (неразборчиво)), она хорошо все сжимает и достаточно безопасна.

Хотел обратить внимание на такой момент, что когда разработчики используют библиотеку JSON для старых браузеров, которая заменяется стандартным JSON, нужно быть очень аккуратными, потому что в некоторых случаях какой-нибудь хакер может с «левого» сайта передать вам не настоящий JSON, а какую-нибудь функцию. Если вы делаете eval что-нибудь вместо того, чтобы парсить JSON, то может быть какой-нибудь хак, функция — и злой хакер получит какие-то данные, доступ.

hl⁺⁺ HighLoad⁺⁺

- Интеграция со сторонними ресурсами
- Twitter

[ВКонтакте](#)

Наверное, последняя тема, о которой я расскажу перед тем, как придет Павел и расскажет, как все было в далеком 2006-м. Интеграция со сторонними сервисами. Как мы строили взаимодействие со всякими Twitter'ами, Share и так далее.

На самом деле, все не так тривиально, как кажется. Мы не можем с нашего рабочего сервера, который принимает запросы, обратиться к Twitter'у. Это плохо по целому ряду причин. Например, есть Coral, который зависает, который оставляет коннекты, стекает память, и сисадмин жалуется.

Есть другие проблемы. Это может быть небезопасно, и вообще сисадмины и так мало спят.

Мы используем примерно такую структуру. Создается большая очередь. Причем специфическая очередь, потому что мы не можем просто взять какую-то очередь. Нам нужно, чтобы все было безопасно. Мы должны понимать, что сисадмины очень часто перезагружают серверы, которые будут выполнять запросы Twitter'у, потому что у них очень часто кончаются коннекты. Лучше не представлять, что там творится. С очередью нам нужно работать очень аккуратно.

hl++ HighLoad++

Интеграция со сторонними ресурсами

- Twitter
- запрос → очередь
- очередь → запрос к Twitter

ВКонтакте

Как это делается. Мы берем небольшую очередь на memcache, небольшую очередь на MySQL (потому что там все может долго продержаться), кидаем туда запрос, что хочет пользователь. Когда наш скрипт (неразборчиво) с отдельных серверов, которые стучатся к Twitter'у, хочет взять что-то с очереди, он говорит: «Я читаю что-то из MySQL». Этот способ очень правильный, потому что если вдруг все упадет, то ничего не будет запрошено второй раз.

У нас везде стоят timestamp'ы, мы знаем, что, во сколько и откуда мы вытащили. Не дай Бог что-то вырубится, все будет хорошо. Memcache очень сильно в этом помогает.

Поставив Memcache lock, сервер снимает данные, начинает их обрабатывать и снимает Memcache lock. Если он не обработает данные, они не вынутся из базы, но при этом их никто не прочитает в течение какого-то времени. Взяв данные, он сразу вписал в них новые timestamp'ы. «Я их взял, поэтому 60 секунд и никто не имеет права трогать».

Через 60 секунд, если у нас что-то не получилось, и он не смог записать, что все это выполнено, кто-то другой возьмет этот запрос и выполнит его до конца.

Что делать, если у нас на сайте обновляется слишком много статусов и довольно приличная аудитория пользуется экспортом Twitter'a. Как быть. В стандартном Cron'e мы не можем поставить что-то чаще, чем на минуту. Но это не проблема. Мы пишем небольшой скрипт, который вызывает это раз в секунду. Он выполняется минуту и раз в секунду вызывает скрипт.



HighLoad++

Интеграция со сторонними ресурсами

- Share
 - запрос → адрес спец. Сервера
 - запрос к спец. Серверу → результат
 - сохранение результата на сервере

[ВКонтакте](#)

Мы можем делать это все более динамично. Мы делаем Ajax-запрос о том, что кто-то хочет что-то зашарить и нам возвращается адрес другого специального load-сервера, который может делать внешние запросы, может обратиться к этому сайту, который хотят зашарить. После этого клиент обращается к этому серверу и получает от него результат, что лежало на этом сайте (такой-то заголовок, такая-то картинка, такая-то информация).

После получения результата он выкладывает его спокойно на сервер, и в итоге ссылка шарится.

Но отдельно хотелось бы рассказать про детали. Раз мы выступаем перед такой аудиторией (очень приятно здесь всех видеть), то хотелось бы анонсировать здесь новую деталь — мы стали поддерживать в Share openGraph теги.

Это перспективное направление, потому что нам нужно знать, что на этой странице, какой контент туда выложили, чтобы адекватно его показать (например, в подсказке при наведении на какую-то ссылку и в других случаях). Нам нужно показать, что там было. Если разработчик сайта укажет у себя в openGraph теге, что это type — какой-нибудь movie, site name — какой-нибудь «КиноПоиск», то в подсказке высветится, что это такой-то фильм с таким-то бюджетом и у него такие-то параметры.

На самом деле, все гораздо сложнее. Мы отвели специальную базу, чтобы хранить объекты openGraph'a. В дальнейшем мы планируем эти данные активно использовать, чтобы помогать пользователям находить ту или иную информацию и проще делать удобные вещи.



HighLoad++

Интеграция со сторонними ресурсами

Open Graph

[ВКонтакте](#)

Еще один интересный момент перед тем, как я закончу, и мы продолжим «круглый стол». Как можно классно выдергивать фотографии, которые относятся к конкретной статье на сайте. Может быть, вы заметили, что в последнее время в Контакте, когда вы что-то шарите, подбирается как раз та фотография, которая подходит к данной статье.

Это делается очень просто. Мы берем alt у фотографии и сравниваем его с заголовком. Если сайт был оптимизирован SEOшниками, то абсолютно точно alt фотографии будет похож на заголовок. Мы считаем количество перестановок букв между заголовком и текстом, необходимое для того, чтобы преобразовать одно слово в другое по словам. В целом получаем схожесть фраз.

На этом, наверное, хотелось бы закончить.

Вопросы и Ответы

Надо сказать, что из этого мы приобрели много плюсов. В итоге мы не тратили кучу времени на интеграцию. Мы очень быстро все интегрировали. Язык позволял это делать действительно быстро. Там все инструменты были.

Не очень много времени мы потратили и на реализацию протоколов. В итоге мы получили некий «велосипед», который немного сомнителен в том плане, что это не готовый веб-сервер, а только некая прослойка. Любой разработчик не может у себя поставить и использовать как полноценный jabber-сервер. Там нет ничего.

Но, с другой стороны, нам не пришлось ничего ниоткуда вырезать, мучаться, как, что, где код. Когда что-то пишется с нуля, то в некоторых случаях тратится меньше времени, чем на использование готовых продуктов, особенно когда от продукта требуется совсем немного и много требуется от того, что нужно написать.

Вопрос из зала: За сколько вы (неразборчиво) node.js с openSSL (неразборчиво)?

Олег Илларионов: Сейчас в той версии, которая есть в репозитории, самая злая проблема следующая. Если, например, у вас много пользователей, все работает, придет какой-то нехороший пользователь и вызовет ошибку в openSSL.

У меня получилось эмулировать ситуацию довольно просто. Я начинал соединение, делал can't shake, а потом посыпал незашифрованные данные. В openSSL возникала ошибка, и в случае этой ошибки все сообщения пользователей, которые будут что-то слать на сервер, будут обрезаться (неразборчиво) с конца до тех пор, пока кто-то новый не подключится и в самой библиотеке openSSL что-то не исправит.

Это решалось очень просто, на самом деле. Как я решил эту проблему. В случае этой ошибки я возвращаю не минус 2, а 0. Ничего нигде не обрезается.

Но я решил, что не стоит коммитить эту ветку, потому что это не совсем правильно. Нужно лезть в код openSSL, смотреть, как он взаимодействует с этими бинингами. К сожалению, на это не было времени.

Вопрос из зала: (неразборчиво) API.

Олег Илларионов: API там очень низкоуровневое.

Вопрос из зала: Нет, вообще.

Олег Илларионов: Относительно API Контакта?

Вопрос из зала: Да. Ситуация в следующем. Недавно было открыто приложение. Сейчас в документации на первой странице: «Для авторизации используйте компонент «браузер». Очень здорово, если мы работаем в X-коде или в Legal Studio. Там либо подтянется движок

Internet Explorer'a, либо сломать там что-нибудь (неразборчиво). Если в нашем каркасе нет браузера, под которым мы хотим разработать десктопное приложение для Контакта, будет ли вводится что-то более приемлемое, например, для авторизации?

Олег Илларионов: Естественно. Много жалоб было от разработчиков Java-приложений для телефонов, где, естественно, даже никакой надежды на браузер нет (в старых телефонах). Мы много думали и собираемся делать open AOS с пин-кодом для таких случаев (пользователю дается пин-код, который он вводит в приложение и авторизуется).

Это будет несколько позже, потому что мы занимаемся тем, что нам нужно настроить HTTPS на всех серверах (это занимает время). Для того чтобы сделать AOS, нам нужен HTTPS.

Вопрос из зала: Теперь Контакт будет доступен по HTTPS, как, допустим, Gmail?

Олег Илларионов: Это еще не факт. Возможно, будет сделано ограничение в динамике и переход на новую версию, потому что фронтов мало, а HTTPS трафик надо обрабатывать на фронтах.

Вопрос из зала: (Неразборчиво реплика).

Олег Илларионов: Я думаю, что на взаимодействии, но не могу сказать точно. Дело в том, что я докопался до С'шного кода, до (неразборчиво). Там эта проблема, возвращаемая от (неразборчиво), присутствовала. Там возвращалась openSSL ошибка, но неизвестно, вследствие чего она была вызвана. Возможно, она была вызвана вследствие неправильной обработки node. Ошибка возникает внутри openSSL, но ее причиной, возможно, стал не openSSL, а неправильная запись в него данных.

Очень сложно дебажить эти случаи, потому что такие ошибки возникают довольно редко и в нестандартных ситуациях (когда пользователи пишут что-то очень неправильное, или у них отрывается коннект в какой-то момент).

Вопрос из зала: Я бы хотел коснуться вопроса garbage collector'a. Он у вас прямо волшебным образом работает, не зависает на несколько секунд или минут. Все нормально справляется со всеми задачами.

Олег Илларионов: Он не зависает, но бывают моменты, когда он вдруг инициируется и собирает мусор. Это может достигать целых ста миллисекунд, но случаев с секундами не было. В принципе, мы не используем принудительную сборку мусора, хотя это возможно в (неразборчиво) 8. Мы экономим ресурсы, хотя мы пытались делать с принудительной сборкой и поняли, что это все-таки не совсем правильно.

Вопрос из зала: Что происходит (неразборчиво)?

Олег Илларионов: Нет.

Вопрос из зала: Какие у вас отношения с Facebook'ом? Вы на них, мягко говоря, похожи. (Смех в зале).

Олег Илларионов: Насколько я понимаю, никаких проблем с Facebook'ом нет. Мне кажется, что глупо, если бы идея социальной сети принадлежала одному сайту, если бы была только одна социальная сеть в мире, и никто больше не пытался бы их делать. Некоторые похожие черты — это тоже понятно. Но надо сказать, что некоторые вещи мы сами придумали, некоторые — сделали первыми. Мне кажется, особенно в этом году это очень хорошо. Мы не стараемся быть похожими на них, но стараемся давать пользователям максимум того, что сейчас можно сделать в области социальных сетей.

Вопрос из зала: Нарисуйте архитектуру.

Олег Илларионов: Чуть позже, сейчас будет «круглый стол».

Вопрос из зала: Вопрос по поводу API. Вы планируете какую-то функцию для работы с группами? В частности интересует загрузка видео через API именно в группы.

Олег Илларионов: Разговоры об этом когда-то были. Правда, не про видео, а про фотографии. В целом если посмотреть на список задач, которые у нас есть, его хватит на несколько лет. Но я думаю, что движения во всех направлениях в области API будут, и особенно это, раз вы сказали об этом здесь.

Вопрос из зала: Вы делаете какое-то нагружочное тестирование перед тем, как что-то выпустить в продашн? Если делаете, то какое.

Олег Илларионов: Смотря чего. Что вас интересует?

Вопрос из зала: Например, та же самая XMPP-информация.

Олег Илларионов: XMPP — конечно. Есть скрипт, который имитирует кучу Олегов Илларионовых, которые коннектятся к серверу. Другие вещи (какие-то продакшн на PHP) тестируются другим способом. Когда выпускаются какие-то сомнительные вещи, которые могут все повалить, выпускаются сначала на первый миллион, потом чуть больше — градацией. Даже иногда сначала на первые 10 тысяч, а потом все больше, больше, больше.

Вопрос из зала: Есть продакшн баз данных. Вы реплицируете в своем тестовом (неразборчиво). Или этим вообще не приходится заниматься?

Олег Илларионов: Когда идет тестирование на пользователях, то не знаю, насколько можно назвать (неразборчиво) случайными, но их забивают пользователи. Первая тысяча, первые 10 тысяч — если мы смотрим, что пользователи слишком сильно груят базу данных, то мы ставим больше кэширования на memcache и так далее. Надо сказать, что не все ВКонтакте работает на MySQL. Многие вещи работают на специальных С'шных движках. Есть небольшая команда С'шников, которые занимаются этим. Особенно нагруженные вещи, естественно, работают на C.

Круглый стол ВКонтакте

Олег Илларионов, Павел Дуров

Олег Бунин: Я рад представить легенду Рунета — Павел Дуров. Все, что вы хотели знать про ВКонтакте, но боялись спросить, кроме одного вопроса (вы знаете какого).



Вопрос из зала: Про стену?

Олег Бунин: Да.

Вопрос из зала: Расскажите, как вы храните пользовательские данные?

Павел Дуров: На дисках. (Смех в зале).

Вопрос из зала: Понятно. Есть какая-то общая файловая система для всех серверов?

Павел Дуров: Фотографии — на файлах обычных. В файловой системе Linux'a.

Вопрос из зала: Это для всех серверов?

Павел Дуров: Разумеется.

Вопрос из зала: А в какой?

Павел Дуров: XFS, естественно.

Вопрос из зала: Как с вашей нагрузкой на сайт вы будете соответствовать закону «О защите персональных данных»? С ГОСТовской стандартизацией, сертификацией и прочим.

Павел Дуров: На 100% будем соответствовать.

Вопрос из зала: Как? Что будете — понятно. Вопрос — как.

Павел Дуров: Я думаю, разберемся. Я просто не могу точно ответить сейчас. Очень много деталей. Есть планы, есть наработки, но сейчас мы углубимся.

Вопрос из зала: Скажите, пожалуйста, как у вас поставлен процесс разработки — сколько разработчиков, менеджеров, кто придумывает идеи, может, используете какую-то методологию?

Павел Дуров: Это близко к Agile. Порядка 30-40 разработчиков. 2 дизайнера. Идеи придумывают 1-2 человека (один из них здесь находится). (*Смех в зале*).

Самое главное: цикл разработки — неделя. За неделю обязательно должен быть готовый результат. Если за неделю мы ничего не видим, готовый продукт не представлен на презентации, то это скорее неуспех, чем успех.



Вопрос из зала: Сколько у вас тестировщиков, нагрузочников, функциональщиков?

Павел Дуров: (*Смеется*). На самом деле, часть разработчиков параллельно функционирует как тестировщики и балансировщики нагрузки. Часть сисадминов занимается тем же самым. Все сисадмины, которые у нас есть, это еще и специалисты по нагрузкам.

Вопрос из зала: Какой процент соотношения нагрузочников и функционала?

Павел Дуров: Порядка 10-20%.

Вопрос из зала: 10% функционала и 90% нагрузки?

Павел Дуров: Нет, наоборот. Порядка 20% специалистов занимается только этим. Параллельно те люди, которые занимаются системным администрированием, следят за нагрузками и пытаются своевременно все проблемы решать.

Вопрос из зала: Сколько человек?

Павел Дуров: Так как у нас многофункциональные специалисты, универсалы, я могу сказать, это порядка 20-25 человек из команды в 40 человек. На самом деле, у нас нет разработчиков, которые не являются специалистами по высоким технологиям.

Вопрос из зала: Что вы используете в качестве инструментария для автотестов?

Павел Дуров: Весь софт свой.

Вопрос из зала: Автотесты тоже?

Павел Дуров: Да.

Вопрос из зала: Автотесты используете? (*Смех, аплодисменты в зале*).

Павел Дуров: У нас поставлены такие интересные собственные разработки, которые мониторят все тысячи серверов. Чуть что где — прсылаются sms (мне, например, постоянно приходят сейчас), и все вопросы решаются оперативно.

Олег Илларионов: Нельзя забыть про целую «касту». В Контакте есть модераторы (люди, которые много писали в техподдержку, становятся модераторами) и так называемые тестировщики. Не могу назвать точную цифру, но около сорока. Перед тем как что-то зарелизить, мы в специальном интерфейсе пишем: «Мы собираемся зарелизить то-то, посмотрите». Они отписывают о багах.

Не знаю, можно ли их назвать тестерами или нет. Это не совсем тестеры, они не сидят в офисе, ничего не тестируют, не исправляют. В то же время они говорят нам о багах, что очень помогает в разработке.

Вопрос из зала: Избыточность как-то реализована на серверах? Пользователи не имеют доступа к данным?

Павел Дуров: Разумеется, избыточность реализована. Во-первых, мы следим за тем, чтобы сервера не работали на 20% CPU, а чтобы они отдыхали большую часть времени. Кроме того, все основные базы данных реплицируются. Все файлы хранятся на двух дисках одного сервера и копируются на старые сервера. Разумеется, ответ «да».

Вопрос из зала: Насколько я понимаю, приватных сообщений у вас очень много (я даже боюсь предположить цифру). Как вы их храните? Какую базу данных используете?

Павел Дуров: Да, это наиболее актуальный вопрос. В день отсылаются 200 миллионов личных сообщений. Для их хранения, обработки была реализована собственная система, свой движок, написанный на языке С. Разработчики, которые им занимались, неоднократные победители олимпиад по программированию, ACM, рейтингов. На самом деле, самые лучшие умы, которые у нас есть, это разработчики, которые поддерживают этот движок.

Это касается, кстати, и сообщений стены, статусов, комментариев к статусам, а также данных, которые касаются поиска. Вся приватность, все списки друзей реализованы с помощью собственной системы хранения данных. При разработке этой системы учитывались малейшие потери в оперативной памяти. Все делалось таким образом, чтобы реализовать максимальную скорость обработки информации, занять место на серверах. По сути, 99% актуальной информации крутится в оперативной памяти, поэтому все происходит достаточно быстро.

Вопрос из зала: Я так понимаю, эта система шардится легко? Без лишних усилий вы ее распространяете?

Павел Дуров: Вы имеете в виду именно систему хранения личных сообщений?

Вопрос из зала: Да, я имею в виду ее, в первую очередь.

Павел Дуров: Да, разумеется, это так. Изначально весь софт проектируется таким образом, чтобы дебрасывание новых серверов не было проблемой. Разумеется, периодически приходится пускать специальные скрипты, которые раскидывают в течение какого-то времени текущие данные на большее количество серверов. Например, у нас было 500 серверов под личные сообщения, стало 700. Какой-то софт в течение пары суток отработает, и будет 700.

Вопрос из зала: Ярослав Городецкий, компания CDMvideo. Есть ли у вас собственная сеть CDN для раздачи горячего статического контента. Где находятся сервера. Будете ли вы (не разборчиво) самостоятельно?

Павел Дуров: Сейчас нет. Мы планируем. Сервера находятся сейчас в четырех data-центрах, строим еще два, так как места будут почти исчерпаны к весне этого года. Они находятся в Москве и в Санкт-Петербурге.

Основная база серверов — в Санкт-Петербурге. В Москве находятся, в основном, видеосервера и немного аудиосерверов. Видеосервера генерируют в пике 160 гигабит в секунду, поэтому из Москвы этот трафик легче отдать.

Вопрос из зала: 160 гигабит в секунду — прикольно.

(Смех в зале).

Вопрос из зала: Можете рассказать о способе реплицирования большой базы данных между московскими и санкт-петербургскими серверами. Или у вас единая база данных в одном data-центре?

Олег Илларионов: База данных, естественно, в одном месте в Санкт-Петербурге. Даже больше того — в одном data-центре. Сейчас как раз ресурсы этого data-центра исчерпаны. Мы работаем над тем, чтобы сделать большую сеть и разрастись на два data-центра (в плане базы данных).

Вопрос из зала: Можете рассказать о принципах реплицирования на большие расстояния, которые вы хотите и будете применять. Это будет какая-то инкрементальная репликация. Как вы будете это делать?

Олег Илларионов: Во-первых, два data-центра, на которых должна быть вся динамика, должны быть сравнительно близко друг к другу. Я думаю, что большая толстая оптика решит проблему долгих репликаций. Не знаю, как это будет устроено. Но я думаю, что люди, которые будут этим заниматься, решат эти проблемы правильно.

Павел Дуров: На самом деле, система готова. Два data-центра в Петербурге, база данных сейчас находится в одном из них. Мы хотим на второй data-центр реплицировать часть. Все испытано, протестировано. Поставим сервера, начнем это делать. Естественно, все написано нашими специалистами. Каких-то готовых решений мы не используем.

Вопрос из зала: Хоть что-нибудь? Nginx, memcached?

Павел Дуров: Это, разумеется, мы используем. Причем по совету Олега Бунина.

Вопрос из зала: Общие детали реализации будут?

Павел Дуров: Реализации чего именно?

Вопрос из зала: Какие вы используете наработки?

Павел Дуров: Да, мы хотели бы вообще этот софт сделать общедоступным, GPL. Он просто сейчас очень специализирован, и нет смысла выкладывать в общий доступ. Мы хотим сделать некую универсальную систему хранения данных — и для себя, и для всех остальных. Представить всем для использования свободно, открыто.

Вопрос из зала: Скажите, пожалуйста, несколько слов об обработке видео. Насколько я понимаю, когда видеофайл загружается на сервер, он каким-то образом анализируется (кодак, формат) и перекодируется в какой-то стандартный вид. Так ли это? Если так, то какие инструменты вы используете. Планируете ли на этом этапе какие-то проверки?... Не лицензионной чистоты, но что-то в этой области.

Павел Дуров: Да-да-да. Здесь мы используем стандартные инструменты, хотя мы их активно дорабатывали.

Вопрос из зала: Какие?

Павел Дуров: Общедоступные. Сейчас работает система, в соответствии с которой повторно загрузить тот файл, который был удален модераторами либо правообладателями, нереально. Берется просто некий хеш от файла, сравнивается.

Вопрос из зала: Есть база этих хешей?

Павел Дуров: Да, разумеется. Мы за этим следим. Постоянно удаляются файлы. По запросу правообладателей мы их удаляем. За этим следят некоммерческие организации, те организации, которые борются с порнографией. Они удаляют. Повторно загрузить это нереально.

Разумеется, если человек отредактирует файл, возьмет какой-то другой кусок и загрузит, это будет работать. Поэтому мы будем совершенствовать эту систему.

Вопрос из зала: Вы и дальше планируете использовать именно FTTP? Не VLC?

Павел Дуров: На самом деле, мы этим не занимались так усиленно. Это не является ядром нашего бизнеса. Я сказал об исходящем видеотрафике. Гигантские расходы.

Основная статья расходов, связанных с трафиком, именно видеотрафик. При этом на страницах с видео у нас нет рекламы, и мы не пытаемся его монетизировать. Ни баннерная, ни таргетированная реклама, ни баннеры — ничего. В том числе, из-за вероятных проблем с правообладателями.

Для нас видео всегда было неким боковым проектом, на котором мы не фокусировались. Недавно мы запустили сервис, который позволяет интегрировать видео с YouTube, с RuTube и так далее. По-моему, Олег как раз интегрировал.

Для нас это очень приятные, позитивные изменения, так как мы не будем нести убытки, связанные с трафиком. Чуть-чуть откладываем проблемы, связанные с построением собственной CDN-сети (Content Delivery Network). Опять же это не ядро нашего бизнеса. Нам не хотелось бы размывать фокус.

Вопрос из зала: В будущем вы, возможно, откажетесь от непосредственного размещения видеофайлов?

Павел Дуров: Скорее, нет, так как это будет ударом по очень многим пользователям, которые не знают про YouTube и другие средства хранения видео. Но это действительно не является чем-то первостепенным. Если мы анализируем поведение юзеров ВКонтакте, мы видим, что 90% их времени уходит на просмотр фотографий, чтение личных сообщений, чтение новостей, и так далее. 10% уходит на потребление всего этого трафика.

Поэтому для нас это не приоритет. Мы не стараемся разрабатывать в данной сфере очень изысканные решения. Я могу сказать об этом открыто.

Вопрос из зала: Вопрос из Facebook. Ребята спрашивают: «Вы сейчас интегрировали YouTube. Планируете ли вы интегрировать другие сети, типа Twitter и так далее?».

Олег Илларионов: Мы готовы интегрировать все, что будет нужно пользователям. Мы занимаемся этим. Мы интегрируемся с Twitter'ом, с другими вещами.

Насчет видеохостингов. Если есть какой-то видеохостинг, который хочет интегрироваться, они могут написать на press@vkontakte.ru и договориться, как это сделал RuTube. Это все обсуждается. Нам несложно подключить какой-то новый видеохостинг. Это не является проблемой, мы с удовольствием это сделаем.

Вопрос из зала: Когда Twitter заработает? Точнее, трансляция.

Олег Илларионов: Трансляция из Twitter в Контакт?

Вопрос из зала: Да.

Олег Илларионов: Это обсуждалось, но пока сложно представить, как это будет выглядеть, как это будет настраиваться и нужно ли это пользователям. Пока это на уровне идеи, и нет четкого представления, как это должно быть реализовано. Поэтому пока не могу ответить на этот вопрос.

Вопрос из зала: Что является ключевым решением для того, чтобы запускать такие процессы? Некая статистика необходимости запросов или это возможность технологического решения и затраты времени?

Олег Илларионов: Как сказал Павел, у нас есть два человека, которые занимаются решением, что нужно сделать. Они анализируют и статистику, и то, сколько нужно затратить времени, и то, какому разработчику это нужно дать (кто лучше справится с этой задачей). На основе этого составляется такой план на неделю.

Вопрос из зала: Некие приоритеты, за которыми следует остальной вопрос.

Олег Илларионов: Да.

Вопрос из зала: Как я понимаю, у вас количество серверов, систем хранения данных считается уже сотнями, если не тысячами. Эти сервера используются в течение многих лет. С течением времени емкость и, в особенности, производительность процессоров, оперативной памяти, систем хранения данных отличается. Как вы учитываете это при балансировке? У вас есть какие-то условные данные, на основе которых распределяется нагрузка?

Павел Дуров: Да, все это так. Абсолютно верно. Данные существуют. Мы знаем, какой сервер с какими характеристиками какую нагрузку может взять на себя.

Вопрос из зала: Для балансировки на back-end'ах это тоже как-то используется? У вас есть какой-то механизм, как у nginx приоритеты по серверам?

Павел Дуров: Да.

Вопрос из зала: Как у вас сделан массовый deploy на ваши back-end'ы: какое-то стандартное решение или что-то самописное?

Павел Дуров: Не могу сказать. По-моему, самописные скрипты.

Олег Илларионов: Ничего такого особенного, никаких пикинговых сетей. Об этом, может быть, были какие-то разговоры, но, в целом, сейчас довольно быстро все deploy'ится. В зависимости от загруженности серверов, но больше, чем минуту эта логика никогда не длилась, поэтому с таким временем deploy'я можно спокойно жить.

Вопрос из зала: Какие примерно мощности сейчас требуются на перекодирование файлов при загрузке (какое количество серверов)?

Павел Дуров: Там не так много — по-моему, 1-1,5 тысячи серверов. (Смех в зале). Это все сервера, которые занимаются хранением видео. Они же параллельно занимаются копированием видео. Часть этих серверов при необходимости задействуется под копирование, после чего они кладут эти файлы на те сервера, которые непосредственно будут их отдавать.

Вопрос из зала: Тогда самый первый вопрос, который у меня был, про размещение картинок и видео. Каким образом веб-сервер получает доступ к конкретному файлу на диске? Всегда сетевая файловая система или они как-то реплицируются?

Павел Дуров: Обычный прокси.

Олег Илларионов: Просто скриптами все делается.

Павел Дуров: Да-да.

Вопрос из зала: По сети?

Олег Илларионов: Да.

Вопрос из зала: То есть файлы — это локальные файлы, по сути?

Олег Илларионов: В локальной сети.

Вопрос из зала: Или локально?

Олег Илларионов: Речь о том, что файл мы отдаем конкретному серверу. Но когда кто-то загружает файл, он может сжаться на одном сервере, а потом быть отправленным на другой. Чтобы задействовать лишние сервера, надо сжать их, потому что сжимается видео многих форматов. Это трудоемкая работа.

Вопрос из зала: Вы говорили об анализе действий пользователей (сколько времени пользователь проводит за чтением и так далее). Как это организовано. Есть ли у вас специальная очередь задач в этой области, так же, как языков пользовательского интерфейса. Как много времени вы тратите на это изучение?

Мужской голос:

Считаетесь ли вы продолжением Империи зла или нет?

Павел Дуров: Коротко — нет. Мы недостаточно времени тратим на анализ логов и поведения пользователей, безусловно. Когда мы встречаемся с коллегами из Яндекса, они недоумевают, почему мы так мало внимания уделяем этому. На самом деле, мы сами недоумеваем.

У нас есть очень простенькие анализаторы траффика, есть простенький собственный сервис статистики. Их мы и используем. Ничего очень глубокого здесь мы не проводим. Это одно из упущений.

Вопрос из зала: Расскажите, как функционирует страничка «Новости». Например, есть какой-то пользователь, у которого есть 10 друзей. Так сложилось, что все друзья разбросаны по всем серверам. Новости, которые генерирует этот пользователь, физически реплицируются на каждый сервер или есть более хитрые методы, как функционирует страничка новостей?

Павел Дуров: Очень интересный вопрос. Мы где-то года полтора делились опытом с коллегами из Facebook того, как кто это делает. Мы были удивлены, насколько похожи аналогичные системы. (Смех в зале).

Причем это вещь, которую, понятно, нельзя скопировать, так как она разрабатывается очень глубоко внутри. Весь этот софт написан на (неразборчиво). Мы не используем для этого MySQL. Естественно, кэшируем данные в memcached.

Все новости дробятся по пользователям именно генератором контента. Пользователь как генератор контента, генератор действия. Существуют сервера-сборщики. На сервера-сборщики идет запрос: «Дайте мне действия этих пользователей». Этот сервер посылает запрос к еще десяти серверам, так как пользователи разбиты по id на ряд серверов. Все эти запросы идут параллельно и параллельно приходят обратно.

За какую-то долю секунды идет запрос от этого сервера-сборщика к десятку серверов и приходят ответы. Все это собирается. Как только приходит последний ответ и формируется финальный результат, он идет наверх, и генерится страничка с новостями.



Вопрос из зала: Кто является инициатором сбора: тот, кто породил событие, или тот, кто читает свою страничку «Новости»?

Павел Дуров: Тот, кто читает.

Вопрос из зала: При генерации новостей у вас используются различные фильтры, а также есть ограничения (например, 50 последних новостей). Эти фильтры участвуют в запросе в базу данных? Или фильтрация происходит в самом конце, когда все данные получены большой пачкой, и после этого вы выбираете только те данные, которые запросил пользователь.

Павел Дуров: Разумеется, участвует.

Олег Илларионов: Только без базы данных. Новости затрагивают базу данных, но в целом там С'шный код.

Вопрос из зала: Когда вы говорите про базу данных, что вы имеете в виду?

Олег Илларионов: Да, по сути, С'шный код — это и есть база данных.

Вопрос из зала: Базы данных как таковой у вас нет?

Олег Илларионов: Нет, у нас есть MySQL, конечно. Но в новостях он участвует в меньшей степени.

Вопрос из зала: Ограничение — последние 50 новостей. Если запрос делается на различные сервера (пользователи находятся на разных серверах), мы заранее не знаем, сколько новостей получим. Как вы поступаете здесь?

Олег Илларионов: Приходит больше новостей — мы отсекаем. Очень много где приходится так поступать. Это касается не только новостей.

Например, надо показать случайных участников в группе. В этом случае мы берем захватом, кэшируем memcached и вытаскиваем только часть. Точнее, вытаскиваем все, отсекаем и отдаем только часть. Так будет быстрее.

Павел Дуров: У нас именно 50 новостей? По-моему, нет.

Олег Илларионов: Да, это, по-моему, давно было. Сейчас больше.

Вопрос из зала: Вы не кэшируете дополнительно уже готовый сформированный список последних новостей? Чтобы если пользователь через 5 минут нажмет F5, снова не запрашивать?

Олег Илларионов: Если честно, я не помню.

(Аплодисменты, смех в зале).

Павел Дуров: Да, мы это делаем. Только не за недавнее время, а за те отрезки времени, которые когда-то происходили (половина, день назад и так далее). Если бы мы кэшировали за недавнее время, тогда бы нам не удалось эффективно группировать. Пользователь загрузил две фотографии, потом еще две фотографии через час. Вы же всегда видите одним блоком, а не отдельными.

Вопрос из зала: Как устроен анкетный поиск? Параметры — город, область...

Павел Дуров: Спасибо за вопрос. На самом деле, это наша гордость. Поиска по такому количеству критериев по пользовательской базе в десятки миллионов человек нигде нет. Там есть такие параметры, как «сейчас online». При этом вы можете посмотреть, кто находится online из людей, которые заканчивали кафедру такого-то факультета такого-то вуза в таком-то году или обитают в таком-то районе или около станции метро. Это достаточно сумасшедшие вещи.

Итак, снова это поиск, который разработан лучшими умами. Я могу назвать этих людей, но...

Вопрос из зала: Назовите.

Павел Дуров: Это чемпионы мира Андрей Лопатин, Николай Дуров. Сейчас к нам присоединились Алексей Левин, Арсений Смирнов, Олег Давыдов, Юрий Петров.

На самом деле, мы совсем не скрываем имена разработчиков, не боимся какой-то утечки информации. Их периодически пытаются хантить ребята из Google, из Яндекса, пытаются их переманить. Однако у нас лучшие условия работы, по крайней мере, в Петербурге точно: и место, и компенсации, и премии, и атмосфера. Я думаю, что можно быть спокойным.

Вопрос из зала: Как я понял, C'шный код, о котором вы сейчас говорите, условно можно называть (неразборчиво)?

Павел Дуров: Да.

Вопрос из зала: И как вы с ним общаетесь?

Павел Дуров: Это нереляционная база данных.

Вопрос из зала: Какая?

(Смех в зале).

Олег Илларионов: Вы спросили, как мы с ними взаимодействуем. Мне кажется, это очень интересный момент. Я узнал об этом на собеседовании какое-то время назад. Меня поразил такой подход, потому что мне бы в голову такое никогда в жизни не пришло.

Очень просто можно общаться с C'шным кодом. Пишется демон, который работает по интерфейсу memcached. Общение с ним идет, как с обычным memcached. Это очень просто. Это интегрируется в PHP, в другие языки и позволяет держать всю систему в каком-то едином формате.

Вопрос из зала: Получается, что у вас действительно сложные выборки хотя бы для тех же новостей. Нельзя же сложные выборки построить на просто неких значениях.

Олег Илларионов: Просто в ключе пишутся специальные символы. Это не SQL, но это специальный ключ, который, естественно, не является аналогом ключа в memcached. По сути, это запрос. Но не SQL. Это не реляционная база данных.

Павел Дуров: У нас нет универсальной системы хранения данных до сих пор. Для новостей пишется отдельный софт, узкоспециализированный, заточенный именно под эти задачи, изначально рассчитанный на определенный род запросов. Не любой запрос туда можно отправить.

Вопрос из зала: Не так давно я прочитал новость, что у вас появилась такая функция, как мобильное позиционирование. Оно, наверняка, должно было работать по координатам базовой станции. Где вы взяли базу данных координат базовых станций и соответствие их кодировкам?

Павел Дуров: У нас нет этой базы данных. Я думаю, что это новость одного из операторов связи.

Вопрос из зала: Нет. У вас наверняка это работает через базовую станцию. Я правильно понимаю?

Павел Дуров: У нас есть сервис геолокаций, который мы запустили в августе.

Вопрос из зала: Он связан с координатами базовых станций, с кодами базовых станций?

Олег Илларионов: Нет, с сотовой связью он никак не связан. Используют GPS в телефоне, в iPhone.

Павел Дуров: Да, используются обычные API, которые предоставляются такими устройствами.

Вопрос из зала: Хорошо. Если хотите связать — обращайтесь.

(Смех в зале).

Вопрос из зала: Главный поставщик серверов для Контакта в Санкт-Петербурге. Расскажите про открытие data-центров, когда.

(Смех в зале).

Павел Дуров: Открываются в Ленинградской области.

Вопрос из зала: Когда?

Павел Дуров: Мы надеемся, весной встретим пару новых data-центров. Один на 5 тысяч серверов, другой на 7 тысяч серверов.

Вопрос из зала: А сейчас их сколько?

Павел Дуров: Более десяти тысяч.

Вопрос из зала: Какие операционные системы вы используете на серверах?

Павел Дуров: Linux.

Вопрос из зала: Везде Linux?

Павел Дуров: Да.

Вопрос из зала: Вы используете какое-то решение или полностью самостоятельно реализовано?

Павел Дуров: Самостоятельно.

Вопрос из зала: Можно подробнее? Серверов-то огромное количество.

Павел Дуров: Я думаю, это серьезный вопрос, на который я в деталях сейчас не отвечу. Есть скрипт, который отсматривает все сервера. Все сервера отсматривают информацию о текущем состоянии. Все это собирается, есть красивый интерфейс у сисадминов. Я сам помогал им его сверстать. При любых дисбалансах сразу присылаются sms.

Там все сделано достаточно брутально. Нет каких-то очень удобных изысков, поэтому мы не беремся все это публиковать в общий доступ для использования. Это очень узкоспециализированные вещи. Нашим сисадминам так оказалось удобно. Они, кстати говоря, очень не любят готовые универсальные решения именно в силу их избыточности. Очень многое все-го, что им нужно.

Вопрос из зала: Сисадмины разработали инструмент для самих себя?

Павел Дуров: Да.

Вопрос из зала: Правда ли, что у них нет как такого нагрузочного тестирования. То есть они все тестируют – выкатывают из одной тысячной пользователей. У вас есть какое-то нагрузочное тестирование?

Олег Илларионов: Абсолютно точно так же, за исключением отдельных вещей. Выкатывается. Вы, наверное, замечали, что в новостях было написано, что первым десяти тысячам – такое-то нововведение.

Вопрос из зала: Часто у вас ломаются, выходят из строя жесткие диски на серверах?

Павел Дуров: Очень часто.

Вопрос из зала: Может быть, какие-нибудь цифры?

Олег Бунин:

10 тысяч серверов. Каждый день – десяток.

Павел Дуров: Порядок такой.

Вопрос из зала: Вопрос насчет тестирования фич на какой-то части юзеров. В каком месте вашей системы (на front-end'ах, back-end'ах или где-то еще) вы разделяете пользователей на «тестовых» (грубо говоря) и продакшн-пользователей и каким образом?

Олег Илларионов: If id меньше...

(Смех в зале).

Павел Дуров: На front-end'е юзеры отсекаются очень редко. Когда надо максимально быстро подняться из состояния, когда сайт упал (у нас была пара таких случаев, когда отключали электричество), memcached, соответственно, обнулился, данных нет. Надо долго-долго восстанавливать. Тогда имеет смысл заниматься чем-то на front-end'ах. Пускать не всех юзеров, так как если они придут все, все снова падает.

Вопрос из зала: Безжалостно каждого второго?

Павел Дуров: Например, да. Еще на front-end'ах мы занимались такого рода деятельностью. Когда были DDoS'ы (весна 2007-го года, например), нам приходилось этим заниматься.

Олег Илларионов: Теперь вопрос от меня. Недавно был доклад про DDoS. Мне хотелось осветить эту тему более подробно, потому что это очень интересно, мне кажется. Каким способом можно бороться с DDoS'ом. Что можно сделать, когда DDoS гигантских масштабов?

Павел Дуров: Честно говоря, не очень люблю вспоминать это время. Нас тогда DDoS'или больше, чем всех остальных в Рунете. Проходящая полоса мусорных пакетов достигала полутора гигабит в секунду в тот момент, когда мы были сайтом на трех серверах. Когда мы стали искать статьи на эту тему в 2007-м году, мы, по сути, ничего не смогли найти. Нам приходилось самим писать и скрипты, которые анализируют логи, Real Time, анализируют запросы, выявляют закономерности регулярными выражениями и перезагружают nginx с новой конфигурацией с (неразборчиво).

Потом это перестало спасать. Надо было работать на уровне switch'ей. Потом и это перестало спасать. Надо было работать на уровне операторов связи. Последнее, правда, было конечной точкой. Нам удалось с этим бороться, но, как я говорю, не очень приятно вспоминать то время. Давайте перейдем к чему-нибудь позитивному.

Вопрос из зала: У вас такое большое количество серверов. Сколько же администраторов их обслуживает и чем они занимаются?

Олег Илларионов: Когда я пришел, было два. Сейчас уже четыре.

Павел Дуров: На самом деле, есть люди, которые находятся в data-центрах. Это дежурные инженеры, которые постоянно меняют сломавшиеся диски и следят за базовым состоянием серверов (за их температурой и так далее). Их много. Они работают по сменам. В любом data-центре они в достаточном количестве.

Есть сисадмины, которые работают у нас, они не находятся в data-центрах непосредственно. Они мониторят всю эту ситуацию. Их было трое, когда Олег (Илларионов) присоединился к нам весной или летом этого года. Сейчас их пятеро. В общем-то, хватает. Это люди, которые не смогли приехать сегодня сюда, так как они очень сильно погружены в мониторинг, во всю эту ситуацию. Пяти человек хватает.

Вопрос из зала: Скажите, что происходит с пользовательскими данными, когда пользователь подает команду на удаление (например, тексты, фотографии)? Удаляются они сразу или через какое-то время.

Павел Дуров: Текстовые данные — зависит от того, что это.

Вопрос из зала: Комментарии.

Павел Дуров: Комментарии удаляются сразу. Они записываются в memcached на 15 минут на тот случай, если пользователь захочет восстановить комментарий. Фотографии удаляются сходным образом, однако сам файл с фотографией не сразу...

Вопрос из зала: Многие заметили, что они по прямым ссылкам доступны еще долгое время.

Павел Дуров: Сам файл по прямой ссылке останется доступным, это правда.

Вопрос из зала: Он будет храниться вечно?

Павел Дуров: Пока не произойдет какая-то работа над этим сервером, зачистка.

Вопрос из зала: Например, я удалил какую-то фотографию полгода назад, она до сих пор доступна по прямой ссылке.

Павел Дуров: Это правильно. Это происходит, чтобы в работе диска не было лишней фрагментации.

Вопрос из зала: А как же приватность?

Павел Дуров: Ссылки на файл удалены ВКонтакте...

Вопрос из зала: Если кто-то перехватил прямую ссылку?

Павел Дуров: Если кто-то перехватил прямую ссылку, то это вопрос не сайта [vkontakte.ru](#).

Вопрос из зала: А к кому же?

Павел Дуров: Вы можете этот файл разместить на любом хостинге.

Вопрос из зала: Нет, любые хостинги удаляют файлы, когда пользователи просят.

Олег Бунин: Нет, это не так.

Павел Дуров: Это не так.

Олег Бунин: Почти все не удаляют. Это нормально.

Павел Дуров: (Пожимает плечами).

Вопрос из зала: Я удалил фотографию, я хочу, чтобы она вообще исчезла.

(Смех в зале, обсуждение).

Вопрос из зала: Если такие тяжелые как модуль Apache и запросы, то значит, надо на множество серверов. Как вы умудряетесь балансировать?

Павел Дуров: Глобальный вопрос.

Вопрос из зала: IPVS, или nginx, или начиная с Round-robin это все идет?

Павел Дуров: Nginx балансирует. При этом есть отдельные сервера, которые исполняют только скрипты, связанные с личными сообщениями, отдельные — только с профилями, отдельные — с новостями и так далее. Соответственно, если из-за каких-то проблем с базой данных или еще с чем-то забиваются Apache, связанные с новостями, то все остальное при этом работает.

Вопрос из зала: А между 32-мя front-end'ами?

Олег Илларионов: Никак. IP'шник. Есть домен, есть NS-сервера, есть IP'шники. 32 IP'шника.

Вопрос из зала: Вчера я задавал такой вопрос Facebook'у и задам сейчас вам. Кто принимает решение о том, что та или иная фича все-таки едет в релиз. Кто несет ответственность за полученное качество продукта?

Олег Бунин: Можно рассказать про пару фич, сдать нам какую-нибудь сенсацию. (Смех в зале). Что нам ждать в ближайшее время?

Павел Дуров: Решения такого рода принимаются мной и еще одним человеком. Его зовут Андрей Рогозин. Это директор по разработке. Он сегодня очень хотел быть здесь, но заболел. Ответственность несем мы и, соответственно, тот разработчик, который это сделал.

Я думаю, тот тренд, который сейчас выходит, это открытие ВКонтакте, интеграция ВКонтакте с другими сайтами, курс на сотрудничество с другими ресурсами. Все это будет развиваться еще год-два минимум, я думаю, и достигнет какого-то максимума.

Вопрос из зала: У вас пять системных администраторов, которые очень заняты мониторингом, наблюдением. Как вы их отпускаете в отпуск, когда они спят, как они дублируют друг друга?

(Смех в зале).

Павел Дуров: Если заниматься арифметикой, то 24 часа в сутках, 5 человек — вполне сносно. Естественно, они координируют отпуска, не уходят в отпуск сразу все. Всегда есть как минимум 3 человека. Делим 24 часа на 3.

Вопрос из зала: Все пять человек взаимозаменяемы?

Павел Дуров: Мы стремимся к этому. Сейчас это не совсем так.

Вопрос из зала: Sms в неслужебное время приходят?

Павел Дуров: Разумеется.

Олег Илларионов: Я даже могу рассказать интересную историю. Однажды целый день мучил одного из наших системных администраторов, потом пошел к товарищу в соседний кабинет, смотрю — он тоже его целый день мучает. Понял, что наши администраторы, на самом деле, многопоточные.

(Смех в зале).

Вопрос из зала: Расскажите самое начало истории.

Павел Дуров: С удовольствием. На самом деле, так как меня интересовали веб-проекты, наверное, с первого курса университета, я откладывал деньги на хостинг. (Смех, аплодисменты слушателей). При этом мне очень помогали Потанинские стипендии (по-моему, их было 3 или 4). Были неплохие суммы — полторы тысячи рублей. (Смех в зале). Президентские, Правительственные стипендии. Набиралось. Я мог оплачивать услуги хостеров.

В итоге к концу 4-го или 5-го курса накопил на выделенный сервер, купил сервер. Я еще подрабатывал, статьи писал. Купил сервер, поставил, оплачивал. ВКонтакте первоначально крутился на этом сервере. Там еще были проекты.

Потом, когда проект развелся до посещаемости в 10 тысяч человек в день и понадобился еще один сервер, то инвесторы заинтересовались очень сильно. Я помню, как мы везли первые два сервера. Сами их устанавливали, потом где-то ночь я пытался устроить взаимосвязь между сервером базы данных и сервером со скриптами.

Вопрос из зала: Они еще в бою?

Павел Дуров: Они хранятся в data-центре. Они не в бою. Директор data-центра говорил, что он хранит их для какого-то музея.

Олег Бунин: Спасибо, Павел.

Архитектура новой почтовой системы Рамблера

Андрей Шетухин

Линейка докладов была выставлена совершенно замечательно. Сначала вышел Володя Габриелян и рассказал, как подбирать команду и как вообще заниматься разработкой сложных систем. Потом наши коллеги рассказали, как сделать сложную систему, очень сложную. А я, пожалуй, попытаюсь рассказать, как сделать так, чтобы почтовая система была максимально простой, насколько это возможно. Ну что ж, начнем.

hl ++ **HighLoad++**

Архитектура новой почты Рамблера

Андрей Шетухин

Итак, пара слов о презентации. Как известно, на этих конференциях архитектуру «Рамблер-Почты» рассказывали, скажем так, два поколения разработчиков. Вообще, Интернет хорош тем, что можно взять какой-то проект и его внешне не сильно менять, менять архитектуру и заниматься очень интересным делом очень-очень много времени. (Смех в аудитории)

Голос с зала: Ну, еще бы.

Андрей Шетухин: Так получается. Вот, поэтому сейчас мы расскажем о четвертой версии «Рамблер-Почты», надеюсь, не последней.

Голос с зала: Раз в год новая конференция — новая архитектура.

Андрей Шетухин: А почему бы и нет. Если все хорошо, то можно сделать и так.

Голос с зала: Нет, если все хорошо, то архитектуру менять не надо.

Андрей Шетухин: Не всегда. Иногда меняются бизнес-требования. Ну ладно, об этом позже.

hl ++ **HighLoad++**

Rambler Mail сегодня

- 240 тысяч новых регистраций в день
- 66 миллионов пользователей
- 20 миллионов "живых" ящиков
- 192 миллиона контактов в адресной книге
- 16 миллионов писем в день

Итак, что такое «Рамблер-Майл» сегодня. Во-первых, это 240 тысяч, примерно три раза в секунду, регистраций пользователей. База у нас сейчас — 66 миллионов пользователей. Это примерно. Понятное дело, что она растет. Из них примерно 20 миллионов живых ящиков.

Вторая после количества ящиков характеристика — 192 миллиона контактов в адресной книге. Это важно тем, что фактически число показывает, что почта живая, активно используется людьми, не только для регистрации на портале, но и для коммуникации с коллегами, сотрудниками и с другими хорошими людьми. Приходит к нам 16 миллионов писем в день. Много это или мало — судить, в общем, вам. Я считаю, что это неплохое количество, и проект можно считать высоконагруженным.

hl⁺⁺ HighLoad⁺⁺

Rambler Mail сегодня

- Мобильная и Wap версии
- Автосборщик почты
- Неограниченное количество почтовых доменов
- 16 видов оформления на любой вкус
- Умная адресная книга
- Иконки сервисов

Что у нас есть — мобильная версия и wap-версии, автосборщик почты — рор3-пылесос, который умеет «высасывать» почту с чужих сервисов, позволяя пользователям иметь одну точку входа, а не лазить по куче почтовых хостингов. На данный момент мы можем хостить неограниченное количество почтовых доменов. То есть сейчас для регистрации открыт только Rambler.ru, домен в зоне Rambler.ru, но мы, в принципе, можем поддерживать сколько угодно доменов без всяких проблем. Из рюшечек и красавиц — это 16 видов оформления на любой вкус. Кто видел новую почту «Рамблер», тот понимает, о чем идет речь. Кто не видел — welcome, посмотрите, может быть, зарегистрируетесь. «Умная» адресная книга с поиском. Ну, и последнее нововведение — это иконки сервисов. «Яндекс» ввел, мы посмотрели, нам тоже понравилось — мы его за два дня сделали.

hl⁺⁺ HighLoad⁺⁺

Программное обеспечение

- FreeBSD 6.7
- MySQL 4.1
- nginx 0.7.X + patches
- Apache 1.3 + mod_perl + PerlXS + C++
- memcached
- Postfix + Dovecot + patches
- Rspamd + Clamav

Работает это на операционной системе FreeBSD. Тут были слова, что кому-то нравится Linux. Вот нам нравится FreeBSD, поэтому у нас сделано на FreeBSD. По-моему, отличный аргумент.

hl⁺⁺ HighLoad⁺⁺

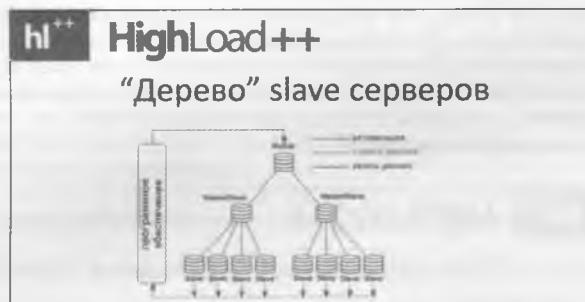
MySQL

- Master-slave репликация
- «Дерево» slave серверов
- Разнесение данных по серверам и таблицам
- Денормализация данных
- Только необходимые индексы в таблице

Второе, что стоит отметить, это MySQL 4.1. Ну, и соответственно, дальше идет некий набор ПО, который мы разрабатывали сами либо дописывали. Это nginx, Apache+mod_perl, memcached, Postfix+Dovecot, как POP3- и IMAP-сервер, Rspamd — новый антиспам, разработанный в «Рамблере», и Clamav для фильтрации вирусов. Rspamd находится в состоянии open source. Если вы зайдете на <http://developes.rambler.ru/>, то вполне можете его скачать и проверить, как он работает. Там уже есть документация. Если у вас есть какие-то проблемы, обязательно обращайтесь. Мы готовы и будем рады вам помочь.

Для чего мы используем MySQL.. В основном, используем MySQL для хранения свойств пользователя, его адресной книги и, авторизационной информации пользователя: как то, логин-пароль, где он находится, в каком из бэкэндов, до недавнего времени — данных для «Рамблер ICQ». В общем, понятное дело, что вся эта штука должна, скажем так, работать «а» - безотказно, «б» - довольно быстро. Что для этого сделано. Есть такое понятие, мы его ввели, как «дерево slave серверов». Следующий слайд покажет, что это такое.

Мы сделали партишенинг для MySQL. Правда, он партишенинг без патча MySQL обошелся, фактически все руками. Но что делать — времени на патч MySQL и доработки сложных систем у нас не было, да и, наверно, это не нужно. Все, что можно, денормализовано, потому что на таких проектах понятие нормальных форм — это такая скользкая вещь, которая, в принципе, хотя и должна быть по науке, а вот на практике почему-то не работает. Ну и такой важный момент стоит отметить, что на всех таблицах мы держим только необходимые индексы. Это азбука, понятно. Но многое проектов, тем не менее, этим азбучным истинам не следуют, и возникают проблемы по нагрузке.



Что такое дерево slave серверов. У нас есть master, от него работает несколько «мастеров», которые являются для верхнего мастера — слэйвом, и мастерами для других нижних слейвов. Пишем мы на мастер, читаем со slave. В реальности схема чуть-чуть сложнее, но выглядит примерно таким образом.

Что такое партишенинг. Стоит отметить, что MySQL — не самая лучшая база данных для хранения больших объемов данных. К сожалению, нам приходится жить с этим наследием, поэтому пришлось провести некоторую работу поパーティонированию таблиц. У нас считается хэш, не обязательно MD5, но для простоты пусть будет так... от которого откусывается кусок какой-то — например, последние три знака. Соответственно первый символ, это номер кластера, остальные два это — номер таблицы в кластере. Таким образом, мы можем получить максимальное количество таблиц 4096 в данной схеме. В принципе, выбирая разное количество цифр хэша, можно получить фактически произвольное количество таблиц, масштабируемое до бесконечности. Если нам надо авторизовать пользователя или попросить его данные, мы знаем его логин, домен и адресуемся к определенной таблице.

hl⁺⁺

HighLoad++

Apache + mod_perl

- OO Perl – хорошо или плохо?
- ORM – зло, которое следует избегать
- SQL Proxy – решение проблем с нагрузкой БД
- Прозрачное кэширование данных
- Быстрый шаблонизатор – СТРР
- Почему не FastCGI?

В качестве сервера бизнес-логики, то есть того, что отдает пользователям почту, нами используется Apache+mod_perl 1.3. Начнем снизу слайд презентации.

Почему не FastCGI? Вообще, FastCGI — прекрасная технология, но практика показывает, что принципиальной разницы между mod_perl и FastCGI нет, потому что основное количество времени — это не взаимодействие между уровнем mod-чего-то там и nginx, или Apache и nginx. А это все взаимодействие — ожидание базы, ожидание кэша... Если у вас есть возможность начинать проект заново, начинайте его с чего хотите. Если есть возможность проект не переписывать или нет желания — не переписывайте, оно будет одинаково работать как на mod_perl, так и на FastCGI.

hl⁺⁺

HighLoad++

OO Perl

- Увеличение скорости разработки
- Сепарация сущностей
- Более прозрачный код в большом проекте
- Медленнее скорость работы
- Выше требования к памяти
- Нужны квалифицированные разработчики



Возникает резонный вопрос: а хорошо ли это — Object-Oriented Perl или не хорошо? Второй вопрос — стоит ли использовать Object-Relational Mapping? Это такая очень модная технология, которая, на наш взгляд, хоть и модная, но слегка не работает. Бывает и такое — не все технологии выстреливают. Из новых разработок есть SQL Proxy, об этом чуть ниже. При нашей нагрузке — необходимо прозрачное кэширование данных, и нормальный вменяемый шаблонизатор (движок для обработки шаблонов).

Итак, ОО Perl — хорошо это или плохо? Это, наверное, философский вопрос, который каждый решает для себя. На наш взгляд ОО Perl, все-таки лучше, чем обычный Perl, которому очень часто люди начинают учиться из процедурных каких-то языков. Для больших проектов это и хорошо, и плохо. Потому что, если у вас есть нормальный квалифицированный работник, написать что-то в ОО-стиле не является проблемой. Это хорошо, быстро, людям интересно, и все такое.

Есть, тем не менее, некоторые «но». Во-первых, это чуть медленнее работает. И в больших проектах это «медленнее» доходит до 20%. Что, вообще, довольно существенно. У вас может быть 100 машин или 120 разница существенна. Каждая машина 3000 долларов — считаем. Выше требования к памяти. Очевидно, что память, конечно, сейчас дешевая, но когда у вас много серверов и очень много процессов на сервере, то, в общем и целом, это дешевизна памяти не совсем и не всегда так.

hi⁺⁺ **HighLoad++**

ORM

- Полный цикл разработки с ORM – дороже
- Простые вещи с ORM делаются просто, сложные – не делаются вовсе
- Много памяти, много кода, много черной магии
- Сложность оптимизации запросов и вообще любой отладки
- ORM – это дорогой вариант sprintf

Почему ORM — это плохо? На самом деле ORM — это такая очень умная, очень быстрая, в теории, очень удобная для пользователей замена sprintf. То есть, мы берем SQL-запрос, в него пишем данные, получаем результат — все хорошо. Это в теории. На практике же получается все несколько наоборот. Вы за 15 минут разрабатываете нечто, что, в принципе, можно показать заказчику или отдать на внутреннее тестирование. Потом у вас уходит несколько недель на то, чтобы прочитать explain неэффективного запроса. Точнее, вытащить запросы из лога, прочитать explain, понять, что ORM несколько не «оэрмет». И вообще, не очень хорошо работает. После чего у вас начинается второй цикл под названием «Вывод приложения на нормальную нагрузку». Иногда это сделать не получается. Потом, есть такая проблема, что простые вещи с ORM делаются, действительно, очень просто: взять Select, поправить ключ, по которому ведется выборка, получить данные — ура, хорошо. Данные запихнули в объект, объект используем. Прекрасно. Как только у нас начинаются вещи чуть более сложные, по крайней мере, в Перле, ORM нормально с этим не справляется. И практика показывает, что программисты берут ORM очередной, а дальше происходит, что? Правильно, половина SQL-запросов — через специальный plaintext интерфейс. Мир, понятное дело, несовершенен. Поэтому обычно на это и списывается, что так получилось сделать.

Кроме всего, обычно важно, чтобы людям было не только интересно что-то писать, а чтобы это дело работало быстро, и, по возможности, с меньшим количеством ошибок. Опять же, практика показывает, что из ORM получается отличный черномагический артефакт, который очень хорошо вертеть в руках, но очень часто из-за него получаются большие проблемы в проекте. Поэтому наша почта не содержит, к сожалению, не единой строки кода никакого ORM-абстрактора, что, в общем, наверное, хорошо.

Что вместо этого есть? Точнее, не вместо этого, а, вместо. Есть инструмент, который позволяет нам сильно уменьшить нагрузку от постоянных соединений. Если вы, например, используете MySQL, у вас есть некая проблема. Вы имеете, к примеру, сто фронтендов. У вас с них идут persistent-сессии к MySQL. Получаем 10-20 child-process в Apache. Каждому процессу необходимо дать persistent сессию. Умножаем. Выясняется, что несколько все не работает как надо. Хотя на самом деле, все эти висящие SQL-сессии, они, по большей части, ничего не делают. То есть, запросы идут относительно нечасто. Пришел пользователь, мы и выяснили, есть ли что-нибудь для него. Какие-то данные, которые надо вытащить из базы. Или надо его авторизовать. А все остальное время подключение висит, по нему передачи данных не происходит. Был разработан софт, который позволяет иметь на входе энное большое количество постоянных подключений от клиентов. А на выходе гораздо меньшее — подключений к базе. Кстати, это тоже будет на <http://developers.rambler.ru/>.

Чем это хорошо? Тем, что фактически к базе идет столько подключений, сколько реально требуется. Если свободных подключений слишком много, лишний thread(нить) убьется, thread убивается — подключение теряется. Если необходимо больше подключений, создаем thread(нить) — подключаемся.

Внешний интерфейс — это JSON. Почему так? Потому что библиотеки, которые обрабатывают JSON, очень много, под любые языки. И не нужно заниматься изобретением собственного протокола, вообще ничего не нужно. Берем JSON.pm, если это Перл, — получаем результат.

hl++ HighLoad++

SQL Proxy

- * Много входящих подключений, мало исходящих
- * Универсальный интерфейс - JSON
- * Простота использования
- * Поддержка Oracle, PostgreSQL, MySQL

Берем библиотеку работы с JSON какую-нибудь, прикручиваем ее к Dovecot или к Postfix, и получаем, результат — мы через свой SQL-прокси гоняем данные безо всяких проблем. Что сейчас оно поддерживает: Oracle, PostgreSQL и MySQL.

Поскольку у нас ситуация такая, что много входящих подключений мультиплексируются в мало исходящих, с транзакциями это не работает. Хорошо это или плохо — решать вам. Наши задачи через эту схему вполне решаются, потому что транзакции почте как-то не совсем нужны.

Кэширование

- * Memcached
- * Драйвер -> Кэш -> Интерфейс
- * Кэшировать объекты или нет?

Есть еще такой вопрос по поводу кэширования. Понятное дело, что используется всем известный Memcached. Есть пара моментов, которые следует отметить. Обычно делается так: берется проект, к проекту прикручивается, например, база данных. Потом проект растет. Потом выясняется, что база данных уже не тянет, ага, давайте прикрутим туда кэш. Прикручиваем снаружи кэш. Что из этого получается? Обычно получаются неприятности. Потому что, когда кэш прикрутили, кто-то должен заниматься его инвалидацией. И если у вас схема чуть-чуть сложнее, чем “пользователь пришел и проапдейтил собственные данные”, то, скорее всего, у вас будет проблема с валидацией. Поэтому в новой почте вопрос взаимодействия с кэшем реализован внутри интерфейсного класса. Грубо говоря, у нас есть класс доступа к адресной книге. Он сам все знает — когда ему надо кэшировать, когда не надо. Когда инвалидировать, когда не инвалидировать.

Стоит ли кэшировать объекты Perl? Как хорошо — берем объект Perl, кэшируем его, получаем результат. Результат плохой. Кэшировать объекты не стоит. Потому что, во-первых, вы в таком случае ограничиваете себя необходимостью использования именно Perl'a. Потому что, если у вас есть объект какой-то, в который вы вносите данные... Что такое объекты Perl'а всем известно. А вы берете его, сериализуете, потом десериализуете. Вдруг вам приспичило все это дело переписать на Perl XS, или на C, для повышения скорости. Тут возникает проблема, что у вас есть куча кэшированных объектов, и непонятно, что с ними делать. Глянцево проще, лучше и интереснее кэшировать не объекты, а их сериализацию типа JSON или PHP-шную сериализацию. То есть, любую такую, до которой, в принципе, можно достучаться из любого языка без существенных проблем. Вывод: кэшировать объекты плохо — потому мы их не кэшируем.



Собственно, вот иллюстрация того, что было сказано. У нас есть некий MySQL Master, MySQL Slave. В принципе, неважно, какая это база данных. Но пусть будет MySQL. Есть некий модуль, который в себя инкапсулирует работу с Мемкэшем. Если у нас есть какой-то mod_perl, он ничего об этом не знает. Когда произошла инвалидация — наше приложение совершенно не волнует. Мы фактически получаем удобно расширяемую, легко поддерживаемую систему.



И все это надо как-то выводить. В некоторых организациях очень любят XML и XSLT. Это очень хорошие технологии, но мы их не используем. Мы используем шаблонизатор CTPP, дополняющий HTML-Template расширениями синтаксиса. Фактически, как и HTML-Template-JIT все из этой серии, как и HTML-Template-Pro. Чем хорош CTPP? Тем, что, во-первых, он разработан не только под Перл, он имеет интерфейс под все распространенные языки программирования. По нашим оценкам этот шаблонизатор является самым быстрым шаблонизатором вообще для Перла. Он быстрее HTML-Template-JIT, который компилирует шаблоны в код на языке С, примерно в 2 раза. Бенчмарки есть, но их можно увидеть на этой вот страничке (http://ctpp.havoc.ru/template_benchmarks.html). Презентация доступна, так что приходите, смотрите.



Хотелось бы рассказать чуть-чуть непосредственно о почте. У нас есть некий набор MX-ов, машины, которые фильтруют спам, фильтруют вирусы. Все это, естественно, пропатчено как раз для того, чтобы работать с SQL-Proxy и сパーティционированными таблицами в MySQL. В планах сделать так, чтобы все это дело работало еще и с Мемкэшем. Тогда наступит счастье, потому что все будет совсем быстро, как только возможно.

hl++ HighLoad++

Postfix + Dovecot

- MX = Mail eXchanger
- Milter API <-> ClamAv + RspamD
- Patches, patches, patches...
- Dovecot
- Хранение писем: Mailbox

Храним мы письма в мейлбоксе. Если мы храним много писем на файловой системе, у нас не кончаются *inodes* и не кончается место намного быстрее, чем нам бы хотелось. Плюс к этому сильно упрощается миграция. Грубо говоря, у нас есть пользователи почтовые ящики, у нас есть бекэнды. Пользователи на бекэндах постоянно растут в объеме. Надо с этим что-то делать. Переносим ровно один файл. После этого пользователь живет на новом бекэнде.

hl++ HighLoad++

Вопросы?



Вопрос из зала: Mailbox стандартный?

Андрей Шетухин: Да. Но есть мнение о том, что его нужно чуть-чуть пропатчить, как водится.

Вопрос из зала: А индекс писем отдельно?

Андрей Шетухин: Надо пропатчить IMAP сервер для того, чтобы у нас был индекс писем, по крайней мере, по всем выводимым нами полям. Это, наверное, будет сделано настолько быстро, насколько это будет возможно. Пожалуй, все. Наверное, получилось довольно долго. Я готов ответить на ваши вопросы.

Вопрос из зала: Вы подготовили схему server-slave. У меня вопрос — запись происходит на master, чтение со slave. А часто такая операция, когда пользователь добавляет какие-то данные, и, обновляя страницу, он видит что его данные добавлены. В таком случае, он же не успеет со slave прочитать?

Андрей Шетухин: А это не требуется, потому что все данные, как правило, после записи оседают в memcached, и на время репликации запись есть в нем, и берется она оттуда. К чему была эта схема с сокрытием Мемкэша — чтобы эти все операции были прозрачны. Чтобы у нас не получалось так, что мы действительно добавили пользователя, между master и slave порядка 7500 секунд разница в репликации, и пользователь видит, что он, вроде, зарегистрировался, а на самом деле его нет. Так быть не может.

Вопрос из зала: Операции у вас не напрямую с базы идут, а через тот интерфейс, через JSON?

Андрей Шетухин: Нет, не все работает через JSON. Но, например, на Перле реализовать это партиционирование довольно просто. Мы считаем хэш, выбираем требуемый коннектор к базе данных — подключились. Если подключение сеансовое — никаких проблем нет. А если же это, например, Postfix, по которому persistent-подключение, длительное, для Dovecot то же самое, то — да, через SQL Proxy. Есть набор патчей, который позволяет Postfix и Dovecot работать с такой кластерной конфигурацией.

То есть, набор патчей довольно простой. Его можно было бы даже выложить. Но с автором Dovecot мы не договорились, а с разработчиками Postfix мы еще не связывались, потому что на это не было времени. Но никаких проблем с тем, чтобы это когда-нибудь сделать, нет.

Вопрос из зала: У меня еще второй вопрос по поводу шардинг. У вас используется статический шардинг, как я понял? Или MD5 от названия ящика?

Андрей Шетухин: Хэш считается, да.

Вопрос из зала: А у вас нет таблиц соответствия? То есть, все определяется по формуле?

Андрей Шетухин: А это не требуется, потому что у нас есть энное количество таблиц, которые сами по себе не нагружены. У них сейчас, несмотря на такое количество пользователей, в среднем по 30-35 тысяч записей. Можете посчитать, простая арифметика: количество пользователей делить на количество записей в таблице — получаем количество таблиц, которое мы используем.

Вопрос из зала: А если будут добавляться новые сервера с новыми базами данных, с новыми таблицами. Как? При равномерном распределении у нас получится, что тут уже есть у нас записи, а на новых серверах пусто. Если мы равномерно будем накладывать, то у нас все не-равномерно получится.

Андрей Шетухин: MySQL-кластер никак не связан с бекэндами, которые являются хранилищами почты. У каждого пользователя в его профайле (properties) прописан почтовый backend, на который он реально ходит. Поэтому мы при заходе пользователя выясняем, какой у него backend.

Вопрос из зала: То есть, у вас это прописано?

Андрей Шетухин: Да. Потом мы идем на этот backend и работаем уже с этим backend-ом. Проблем нет никаких, потому что регистрация выбирает бекэнды по определенному алгоритму. Он недалеко ушел от round-robin с весовыми коэффициентами.

Вопрос из зала: То есть, у вас все-таки не статический, а динамический шардинг получается?

Андрей Шетухин: Почты — да. Истории записей (протокол изменения свойств пользователя) — нет.

Вопрос из зала: Доклад называется «Архитектура новой почтовой системы «Рамблера». Какая там проблематика? То есть, какие задачи решались? Я понял, что вы сильнее сделали свой прокси. Потому что...

Андрей Шетухин: Я считаю, что 200 миллионов записей для таблицы БД MySQL — это преступно. И, соответственно, это подлежало искоренению как можно быстрее.

Вопрос из зала: Есть, например, софт, о котором вы говорите? Вы написали новый, потому что он быстрее?

Андрей Шетухин: Потому что он работает по-другому.

Вопрос из зала: А шаблонизатор вы меняли, потому что у вас свой интерфейс?

Андрей Шетухин: Шаблонизатор меняли по той причине, что предыдущий HTML-Template-JIT давал нам минимальный квант выкатки — одни сутки. Что было недопустимо. Сейчас у нас

выкатка идет 15 минут. И я считаю, что это очень не плохо с точки зрения скорости исправления багов, и, вообще, скорости релиза новой версии.

Вопрос из зала: Какие у вас задачи интересные стоят в ближайшее время?

Андрей Шетухин: Отличный вопрос. Я даже не знаю, как на него ответить. Если я отвечу полностью — меня уволят. А если я отвечу не полностью, то, наверное, это будет не очень честно. Пожалуй, самым лучшим ответом будет — следите за пресс-релизами. Очень скоро они будут довольно интересными.

Вопрос из зала: Вы сказали, почему FreeBSD, но не сказали — почему MySQL.

Андрей Шетухин: MySQL — так вышло. А чем его заменять? Тут вопрос очень простой. Во-первых, зачем менять что-то, что путем небольших переделок все-таки работает? Плюс минус, но работает. Во-вторых, есть такой момент, «ну, давайте его на что-нибудь заменим». Но замена должна давать нам некий набор профита, который оправдывает эту замену. Есть же какие-то затраты на внедрение, плюс миграция пользователей, плюс возможные простоя, плюс потеря какого-то количества пользователей из-за того, что они не захотят ждать миграцию. Вдруг что-то упадет. Все это довольно дорого стоит. Если есть вариант сделать что-то дешево и быстро, и чтобы это работало, то, наверное, это и надо делать так.

Вопрос из зала: Правильно ли я понял, что этот SQL-proxy — это вы изобрели для MySQL pool-соединений, который еще и транзакции не поддерживает?

Андрей Шетухин: Транзакции он не поддерживает. Но, еще раз говорю: нам это не надо. Во-вторых, да, мы смотрели на предмет того, что уже есть. Но на dev.mysql.com есть подобный же проект, который находится в стадии альфа, и не работает с 4.1. И написано, что он не для продакшн. Наверное, вопрос о его использовании как-то не стоит. Поэтому пришлось изобретать что-то свое, чтобы, по крайней мере, работало.

Вопрос из зала: Это просто к вопросу, почему MySQL. Потому что в различных системах pool-соединений уже из коробки есть.

Андрей Шетухин: А у нас нет этих различных систем.

Вопрос из зала: Вопрос по поводу ORM. Вы много невероятных слов в адрес этой технологии сказали. Я хотел спросить, это опыт печального промышленного использования или это просто мнение группы разработчиков?

Андрей Шетухин: Это опыт печального промышленного использования. Фактически мы получаем некий набор ПО, который, еще раз повторю, является заменой sprintf. Кроме этого ничего не делает. При этом жутко требователен к ресурсам и к квалификации специалистов, потому что достаточно одного неправильного хода, и у вас получаются неприятные проблемы. Есть пример, почему в кривых руках ORM плох. В одном из проектов был совершенно замечательный ORM... Был один проект, где сначала для выгрузки, к примеру, десяти тысяч значений делалась следующая вещь. Выделялось 10 тысяч primary key из таблицы, и на каждый вызов итератора iterator.next делался еще один SQL-запрос для выборки непосредственно данных. Итого у нас получилось десять тысяч и один запрос. Понятно, что все это исправляется, и в «прямых» руках все это дело очень легко служит.

Вопрос из зала: Те же самые ошибки можно допустить и без использования ORM.

Андрей Шетухин: Гораздо сложнее, потому что наличие SQL-запроса в plaintext-виде дает возможность делать explain и наказать за то, что он неправильно написан. С ORM иногда получается такой момент — есть такое специфическое заболевание у программистов, оно называется «за меня все придумали, и я использую по мануалу, и потому у меня нет проблем». На самом деле, проблемы есть — либо мануал не прочитан, либо в мануале неправильно написано.

Вопрос из зала: Если хороший разработчик, ему ORM не нужен, он и так с базой данных общается. А если плохой, то...

Андрей Шетухин: Мое личное мнение, в подобных проектах в ORM необходимости нет вообще, потому что простые вещи он делает относительно просто, но сложные не делает практически никак. Либо это делается при помощи «допилки» и разбирательства. Зачем это нужно?

Вопрос из зала: Вы используете Мейлбокс, а не Мейлdir. Насколько я знаю, у Мейлбокса большие проблемы — когда он распухает, там становится много писем. Вы как-нибудь его «пилите», я не знаю?

Андрей Шетухин: Во-первых, все пользователи не вечно. (Смех в зале, аплодисменты.) Некоторые пользователи вечно. И некоторые пользователи, особенно, которые долго куда-то к нам не ходят, подвергаются жесткой процедуре миграции. Потому что другие пользователи на бекэнде растут, нужно как-то получать свободное место, и это свободное место ниоткуда не берется. Оно берется только при помощи миграции неактивных пользователей. Соответственно, такая проблема, как распухание и фрагментация почтового ящика, не особо существенна — рано или поздно все пользователи будут перенесены.

Вопрос из зала: У вас Rspamd занимается только статическим — «пилит» на слова и словарь составляет? Или как?

Андрей Шетухин: По поводу Rspamd. Павел Рогожин меня очень сильно попросил сделать так, чтобы на developers.rambler.ru пришла куча народа. Поэтому я ничего на него отвечать не буду, на этот вопрос. Зайдите туда, прочтите — там есть документация.

Вопрос из зала: Почему MySQL 4-я, а не 5-ка?

Андрей Шетухин: Потому что переход на него, очевидно, будет сложен и не особо необходим.

Вопрос из зала: Вы сказали, что у вас письма хранятся в Мейлбоксе. Какая информация, оставшаяся, хранится в MySQL?

Андрей Шетухин: В MySQL хранится фактически имя бекэнда, на котором пользователь живет, и логин-пароль его. Собственно, — все. На самом деле, хранится там гораздо больше, но для доступа к почте больше не нужно.

Вопрос из зала: Список писем последних — это фактически...

Андрей Шетухин: Список писем, как правило, хранится в Мемкэше.

Голос из зала: Вы упомянули про неограниченное количество почтовых доменов. Сейчас на «Рамблере» можно зарегистрировать красивый домен?

Андрей Шетухин: Сейчас нельзя. На данный момент нельзя. Но, еще раз, почта «Рамблера» спроектирована так, что количество поддерживаемых доменов принципиально не ограничено ничем. Мы можем поддерживать сколько угодно и кого угодно.

Вопрос из зала: Это в плане авторизации, что ли мультидомен?

Андрей Шетухин: Это в плане авторизации почтового хостинга.

Вопрос из зала? В DNS-сервере вы еще допишете одно?

Андрей Шетухин: Зачем? Вообще-то проблема иных записей в зоне — это проблема владельца зоны.

Секция «Системы хранения»

Почему не стоит использовать MongoDB

Сергей Тулунцев



Почему не MongoDB

Сергей Тулунцев

Сергей Тулунцев: Сегодня я вам расскажу про MongoDB. MongoDB — это хорошая база данных. Многие ее любят, используют везде, где только можно, и даже иногда там, где нельзя. Попробуем сегодня рассказать, почему вы можете не захотеть использовать ее для своего следующего проекта. Сразу предупреждаю, что причин, на самом деле, не очень много и не все из них серьезные.



HighLoad++

MapReduce

- Медленный
- Однопоточный

Начнем с MapReduce. Как известно, MongoDB относится к классу так называемых NoSQL-решений. Соответственно, в нем нет SQL и соответствующих возможностей по обработке данных. Есть свой язык запросов с какими-то возможностями. Если этих возможностей не хватает, то предлагается использовать MapReduce.

Вообще это механизм мощный, многие его используют. Но конкретно в этой реализации есть два фатальных недостатка. Первый — медленный, очень медленный. Второй — он однопоточный.

Оба эти недостатка обусловлены тем, что функции мы пишем на JavaScript'е, и выполняется это тоже внутри встроенного JavaScript движка. Термин «однопоточный» в данном случае означает то, что в каждый конкретный момент исполняется только один экземпляр JavaScript-кода. Но экземпляров, запущенных MapReduce, может быть несколько. Чтобы погодробнее понять, почему это медленно и не очень подходит для Real Time запросов, рассмотрим подробнее алгоритм работы.

hl ++ HighLoad++

Map

- Читаем входную коллекцию (read lock)
- map для каждого документа (JS lock)
- Пишем во временную коллекцию (write lock)

Сначала у нас идет стадия mapping'a, мы читаем входную коллекцию. Для этого мы берем read lock и отпускаем каждые 100 документов для того, чтобы дать другим операциям возможность выполниться.

Далее, как мы считали эту пачку документов, мы для каждого документа выполняем map. Для этого мы берем JavaScript lock и преобразуем документ из BSON в JSON. Если функция map у нас выполнила лимит, и нам нужно записать что-то во временную коллекцию, мы пишем это во временную коллекцию. Для этого мы берем write lock.

hl ++ HighLoad++

Reduce

- Читаем временную коллекцию (read lock)
- reduce для каждого документа (JS lock)

Итак, когда-нибудь это закончится, и наступит стадия Reduce. Здесь мы делаем все то же самое, только берем временную коллекцию в качестве входной и функцию reduce как функцию, которую надо выполнить над документом. Однако результаты, которые возвратит эта функция, мы не записываем никуда. Мы пока накапливаем их в памяти.

hl ++ HighLoad++

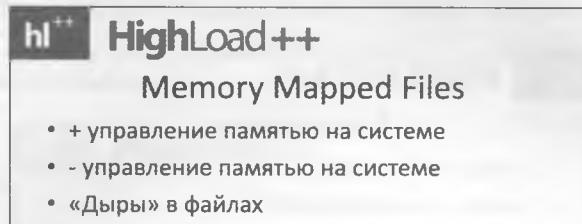
Пост-обработка

- Пишем в выходную коллекцию (write lock)

Потом наступает момент пост-обработки, и мы их все разом записываем в коллекцию, которую мы указали как выходную. Для этого мы берем write lock и записываем все их пачкой,

а не каждый документ по отдельности. Все эти мелкие lock'и и JavaScript, преобразование из BSON в JSON не очень положительно влияют на быстродействие.

Ради теста я погонял на таблице в миллион документов. Одни и те же результаты посчитал встроенным языком,строенными средствами и MapReduce. MapReduce был в 4 раза медленнее при условии, что это был единственный запрос в системе, другие запросы не выполнялись.



Memory Mapped Files — это механизм современных операционных систем, который позволяет установить прямое соответствие между файлом на диске и областью памяти. После этого приложение может работать напрямую с памятью, читать и писать в нее. Система позабочится о сбросе изменившихся участков на диск. В приложении в результате увеличивается быстродействие, потому что доступ к памяти на порядки быстрее, чем системный вызов.

Но есть и недостатки. Недостаток ровно тот же — управление памятью находится на системе. Мы не контролируем, какие страницы у нас находятся в памяти, какие вытеснены на диск.

Конкретно в MongoDB очень важно, чтобы в памяти были хотя бы индексы, которые мы используем при запросах. Если индексов в памяти нет, будет очень плохо.

Допустим, что у нас есть система, она как-то работает. Кто-то запускает массивный MapReduce на большой коллекции, и он начинает шерстить эту коллекцию, поднимать все в память, вытесняя индексы. При том, что сама эта операция будет небыстро работать, потому что, в основном, читает с диска. Остальные запросы у нас тоже начнут тормозить. Система может очень долго восстанавливаться после такого запроса.

Если бы мы управляли памятью сами, то теоретически мы могли бы лучше знать, какие области у нас горячие, какие не надо выгружать или, например, ограничить размер буфера для таких full-scan операций.

Следующий недостаток — это «дыры» в файлах. Когда мы удаляем какой-то документ, он, на самом деле, конечно, физически не удаляется. Он помечается как «удаленный» и будет совсем удален при следующей дефрагментации коллекции. Но до тех пор он будет лежать на диске и занимать место.

Представим себе такой экстремальный случай, что мы удалили большинство документов, но равномерно (4 из каждого пятого). Мы так это дело и оставили, пытаемся посчитать какую-то агрегатную статистику. В результате, чтобы загрузить 5 документов, нам потребуется загрузить в 5 раз больше страниц. Если раньше они умещались, скажем, на одной странице, то теперь на каждой странице будет лежать один реальный документ и 4 фантомных. Это лишняя нагрузка на диск и расход памяти. Конечно, после таких массивных операций надо делать compact.

Во второй версии MongoDB, которая вышла недели 2-3 назад, появилась команда compact, которая позволяет дефрагментировать отдельную коллекцию без необходимости делать это для всей базы данных. Но такая операция по-прежнему блокирует все остальные операции, поэтому лучше производить ее на slave'e.

Еще один недостаток, связанный с памятью, но напрямую не связанный с технологией MMF. Это то, что в MongoDB нельзя ограничить размер используемой памяти. Ребята из «Selectel» (хостинговая компания) утверждают, что MongoDB читает объем доступной оперативки при старте и не проверяет потом это значение. Соответственно, это создает проблемы при запуске MongoDB в облачной среде, где объем доступной оперативки может как повышаться, так и понижаться.

Мне захотелось проверить эту информацию. Я задал этот вопрос на официальном форуме, но никто из «10gen» мне не ответил. Поэтому предлагаю поверить ребятам из «Selectel».

HighLoad++

Блокировки

- Глобальный write lock
- Write lock yielding (new)

Одна из самых больших проблем MongoDB — это блокировки, гранулярность блокировок. В моем личном чарте это вообще первое место. В MongoDB принята модель «1 писатель и много читателей». Причем писатель один не на документ или коллекцию, как во многих других БД, и даже не на базу данных. Он один на сервер. Сколько бы у нас баз данных на сервере ни было, только в одну коллекцию в данный момент может прийти запись (В версии 2.2 была уменьшена гранулярность блокировок. Теперь она ограничена базой данных, а не всем сервером).

Причем этот писатель блокирует всех остальных (других и писателей, и читателей), и один неудачный массовый апдейт может положить всю систему. Он клал всю систему до версии 2.0, в которой он научились ненадолго отпускать write lock при таких массовых апдейтах и, таким образом, давать шанс другим операциям выполниться. Как это работает на деле, я, честно говоря, не проверял. Но обещают, что оно так работает.

Сюда же, к блокировке относим блокировки с миграцией чанков. Когда кластер, шарды и все такое. Вкратце поясню процесс миграции чанков. Когда надо мигрировать часть данных на какой-то новый шард, выбирать этот новый шард (наименее загруженный по умолчанию), данные копируются туда, происходит передача прав этому новому шарду путем записи в config сервер, и данные удаляются со старого шарда.

Так вот, в версии 1.6 жутко тормозило это удаление. Оно могло висеть часа 3 спокойно и блокировало все остальные операции. Я достал всех разработчиков по этому поводу. Они что-то там подкрутили, и перестало тормозить.

Потом вышла версия 1.8, в которой как раз это перестало тормозить. Но миграции продолжали зависать. Полез копать дальше — выяснилось, что во время миграции чанков происходит вызов некоей процедуры setShardVersion, которая, видимо, изменяет что-то в config серверах. Она вызывается несколько раз за миграцию. Она тоже повисала очень легко минут на 20. Иногда отрабатывает, иногда повисает. Лечилось это перезапуском сервера, потому что ждать, пока отвиснет, очень не хотелось.

HighLoad++

Оптимизатор запросов

- Только один индекс
- Эмпирический

Запросы и оптимизация. В настоящее время MongoDB может использовать только один индекс при исполнении запросов, даже если этих индексов есть несколько. Соответственно, это отмечает всякие продвинутые оптимизации вроде index merge. При этом он индекс не всегда угадывает правильно с точки зрения разработчика. Разработчик думает, что должен использовать один индекс, он использует другой. Почему — потому что он выбирает этот план эмпирически.

В первый раз он тупо запускает все доступные планы запросов и смотрит, какой выполнится первым. Этот он считает оптимальным и будет в дальнейшем использовать его. Через некоторое время он повторяет эту попытку. Снова запускается план, он смотрит, какой выполнится быстрее, потому что данные в коллекции могут измениться таким образом, что предыдущий план уже не является оптимальным.

Шардинг

- Все шарды равноправны
- Нет распределения коллекций

Шардинг. В MongoDB, как во многих других системах, шарды равноправны. Данные на них распределяются равномерно. Однажды, еще находясь под эйфорией и будучи излишне оптимистичным по поводу сообразительности MongoDB, я добавил новый шард в кластер. Все было ничего, но этот шард был в 3 раза меньше текущих машин, которые у меня были. Шард добавился, данные начали раскидываться, я пошел спать. Утром проснулся — данные перераспределились, и все тормозит. Работает, но тормозит, очень тяжело отвечает. На этот шард распределилась треть данных. Было два шарда, я добавил еще один, соответственно, везде стало по трети. Я наивно думал, что на него положится одна седьмая. Но нет, все начало тормозить, и тормозило еще два дня, пока я изымал этот шард из кластера.

Здесь ошибка, конечно, моя. Но, кажется, было бы неплохо уметь задавать вес шарда, потому что не всегда удается достать одинаковые машины. В config'e задавать вес: этот может хранить X данных, этот — 2X, этот — 0,3X. Тикет завел — пока ничего непонятно.

Дальше идет объективный недостаток — то, что у нас нет распределения коллекций. Когда мы в кластере MongoDB создаем новую базу данных, она размещается на наименее загруженном шарде. Это считается домашним шардом. После этого все создаваемые коллекции будут размещаться на этом домашнем шарде. Данные могут быть распределены на другие шарды только в том случае, если мы включим шардинг для коллекций.

Включать для маленьких коллекций — это себе дороже выйдет, лучше не стоит. Если у нас много небольших коллекций, то мы можем серьезно с этим попасть. Например, мы храним какие-то данные, логируем данные и группируем их по дням. Один день — одна коллекция. Для таких коллекций включать шардинг лучше не стоит, но и на одном шарде они все не поместятся. Для таких случаев либо не использовать кластер MongoDB и шардить все вручную, либо использовать более традиционные средства.

Мониторинг

- Нет New Relic RPM
- Есть MMS

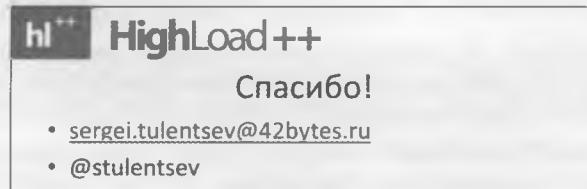
Мониторинг. Здесь просто. Для каждого приложения важно его мониторить. Я конкретно пользуюсь сервисом, который называется New Relic RPM. Он мониторит приложения, производительность и собирает статистику, возникающие ошибки и многое другого. В частности, он мониторит работу с базой данных. Потом можно посмотреть в «админке», какие запросы к базе данных пользуются популярностью, какую нагрузку создают, среднее время выполнения — много чего.

Если какой-то запрос к базе данных у нас занимает неприлично большое количество времени, по нему можно посмотреть дополнительную информацию. Какие конкретно запросы к базе данных были отправлены, сколько времени заняли, можно посмотреть конкретные подставленные туда параметры, план запроса — все, что в голову придет.

Так вот, ничего этого для MongoDB нет. Как объяснили ребята New Relic, это обусловлено ее курсорной моделью работы. Впрочем, какие-то крупицы информации собрать можно. Ну, на уровне, что «во время этого запроса к веб-серверу у класса «юзер» 3 раза был вызван метод `find`». Инструментируется конкретный фреймворк доступа к данным, а не работа с базой на более низком уровне. Для тех, кто привык к подобной отчетности, это может стать серьезным недостатком. Для меня это является серьезным недостатком, но почему-то я пока еще использую MongoDB.

Второй пункт. На самом деле, не недостаток. Просто мне лень было заводить для него отдельный слайд. Перед конференцией на прошлой неделе в четверг компания «10dep» анонсировала свой сервис мониторинга MongoDB (сокращенно MMS). Бесплатный сервис.

Скачивается агент, устанавливается на сервер, настраиваются хосты и снимается статистика по MongoDB. То же самое, что вы могли смотреть через Mongostat, что-то выцеплять в логах. Теперь это централизованно собирается и отправляется на сервис, где это можно смотреть в виде красивых графиков. Если на машине стоит Munin, то он будет подцеплять данные еще из него.



Собственно, вкратце все, что я хотел вам рассказать. Спасибо за внимание. Вопросы, если есть.

Вопросы и Ответы

Вопрос из зала: Можно вопрос про New Relic?

Сергей Тулентев: Да, конечно.

Вопрос из зала: Если вам так хотелось в New Relic увидеть MongoDB, то почему не взять и дописать? Там 20 строчек. По примеру Memcached — два дня работы.

Сергей Тулентев: В том-то и дело, что такая инструментация уже написана. Там даже строчек 10, но она инструментирует конкретный маппер (`Mongoid`, `MongoMapper` и все, по-моему). Конкретно драйвер она не инструментирует. Как сказали ребята из New Relic (я просто зывал, что я к ним обращался и писал им еще раз: «Почему у вас нет инструментации?»), объясняли, что можно это сделать, но это будет очень дорого с точки зрения производительности или памяти. Это будет тормозить клиентские приложения, чего очень не хочется.

Вопрос из зала: Я просто удивлюсь, если это будет дороже, чем инспектирование Memcached. Для Memcached они написали, тут абсолютно аналогичная ситуация.

Сергей Туленцев: Нет, а что Memcached? Там Key-Value: запрос — ответ. Ответ приходит сразу. В «реляционках», в принципе, то же самое. MongoDB — курсоры. Вы отправили запрос, получили пачку данных, потом эти данные можете долго сосать. Сразу собрать по этому запросу полную статистику, видимо, получается очень неудобно.

Вопрос из зала: По ощущениям — на какой объем данных можно замахиваться с MongoDB, а на какой страшно?

Сергей Туленцев: Все эти страшные истории, которые я тут рассказывал. У меня была база гигабайт на 500. В принципе, она держала неплохие нагрузки. Ну, средненькие нагрузки. Тысяч 7 запросов в минуту. По здешним меркам, наверное, копейки. Но такие держала.

Вопрос из зала: 7 тысяч — это со скольких серверов?

Сергей Туленцев: Два шарда.

Вопрос из зала: Насколько я знаю, по дефолту MongoDB не кидает ответы на записи в базах. По-моему, это большая проблема.

Сергей Туленцев: Если вам нужны ответы, подтверждение того, что он сделал fsync, это можно запросить при отправке insert'a.

Вопрос из зала: Учитывая высказанные недостатки, в каких сценариях стоит использовать MongoDB, в каких не стоит?

Сергей Туленцев: Я ее сейчас использую в сценарии, где количество данных не очень быстро растет, но происходит очень много мелких апдейтов, по сути, в горячие данные. На диск они не пишутся. Они постоянно обновляются в память и потихоньку скидываются на диск. Все это довольно быстро работает. Выдает тысяч 30 запросов в секунду.

Вопрос из зала: Спасибо за то, что с нами поделились вашей болью. Почему вы до сих пор используете это?

Сергей Туленцев: Подкупает отсутствие схемы и быстрота разработки. При каждом новом проекте возникает ситуация, что опять будет тормозить, что-нибудь еще. Но как только представляю себе, что надо будет писать миграции для классов, продумывать заранее структуру... Нет, не могу (смеется). (Смех в зале). Очень подкупает.

Спасибо!

12 вариантов использования Redis — в Tarantool

Александр Календарев, Константин Осипов

hl⁺⁺ HighLoad++

**12 примеров
использования Redis
- в Tarantool**

Александр Календарев, Константин Осипов

Константин Осипов: Доброе утро! Меня зовут Костя Осипов. Я разрабатываю систему... Я хотел бы понять, насколько аудитория знакома с Tarantool'ом. Мы собираемся сравнивать его с Redis. В принципе, архитектура похожа, но не совсем.

Дайте мне представить себя и моего коллегу по докладу. Это Александр Календарев. Александр использует Tarantool в одном из своих проектов (он сам из Питера). Я, соответственно, Tarantool разрабатываю.

Прежде чем мы приступим, давайте узнаем, кто вообще представляет себе, что это такое. (*Вопрос к слушателям*). Кто работал с Redis до этого? (*В зале поднимают руки*). О, ну, здесь вообще все пользователи Redis. Кто знает, как в Redis осуществляется многозадачность, как Redis переключается между разными соединениями, обслуживает разные соединения? (*В зале поднимают руки*). Людей гораздо меньше. Кто знает, как Redis хранит данные на диске? (*В зале поднимают руки*). Тоже совсем немного человек.

Давайте тогда, прежде чем мы начнем, я немного расскажу про Redis и Tarantool в сравнении. Архитектуры во многом похожи, но мы уделяем внимание каким-то моментам. Redis считается Data Structure Server. Он хранит в памяти структуры данных и дает к ним доступ через сеть. При этом можно с помощью снапшота сохранить данные на диск. У Redis есть такая возможность, как Append Only File (то, что в Tarantool называется Write-Ahead Log, в этих двух названиях есть разница семантическая), которую можно включить, и тогда все изменения будут также попадать в этот файл. У вас будут сохраняться ваши изменения примерно с той же скоростью, с какой они происходят.

Tarantool и Redis также используют кооперативную многозадачность. Кооперативная многозадачность — это относительно современный подход. Хотя даже MySQL в 2000-м году использовала *Meet on Threads*. Это была такая кооперативная многозадачность. Современный подход к многозадачности, когда используется максимум один процессор. При этом параллельные потоки в программе кооперативно (то есть друг другу) передают управление.

Какие преимущества этого подхода? Это lock-free подход. Не нужны ни mutex'ы, никакие точки синхронизации, потому данные общие, в один момент времени с ними работает один процессор. Соответственно, все издержки таких баз данных, как Postgres, MySQL, вообще любых многопоточных и многопроцессных баз данных здесь уходят. Эта модель похожа на модель Erlang'a, когда вместо того, чтобы использовать общую память, узлы обмениваются между собой сообщениями. Это другая модель данных.

Я, наверное, использовал те 5 минут, которые у нас были. (*Вопрос к залу*). У кого есть вопросы по тому, что я сказал? Нет? Ничего непонятно?

Вопрос из зала: Правильно ли я понимаю, что один инстанс Tarantool может использовать только один core?

Константин Осипов: Правильно ли вы поняли, что один инстанс Tarantool использует один core. Совершенно верно. Типичная конфигурация — это то, что на одной машине работает два-четыре инстанса. Просто за счет того, что вы все равно шардите ваши данные, вы разбиваете их между несколькими инстансами на одном компьютере.

Вопрос из зала: Не совсем понятно по поводу mutex'ов. Вы говорите, что их не нужно использовать вообще, получается?

Константин Осипов: Да, Tarantool и Redis не используют mutex'ов вообще. Синхронизация как таковая не нужна. Все, что нужно выполнить атомарно, просто выполняется атомарно. Управление другому coroutine, другому зеленому потоку не отдается до тех пор, пока эта критическая секция не завершена. В mutex'ах нет необходимости.

Александр Календарев: Я хотел просто добавить. Вчера был очень похожий доклад во второй секции, предпоследний. Там рассказывали про фреймворки, HTTP-сервер (забыл сейчас точное название). Там как раз была похожая технология. Используется libcore. Там на одном потоке используются как раз эти параллельные вычисления.

Вопрос из зала: Почему название ядовитое?

Константин Осипов: Почему ядовитое название. К сожалению, этот вопрос преследует меня, как чума. Название придумали (я не буду говорить «воспаленные») усталые умы программистов, поэтому не судите строго. В принципе, это «tool». Если вам не нравится слово «tarantool», можете говорить, что это «tnttool». Это некий инструмент.

Название отражает модульную структуру сервера. Есть некий фреймворк для работы с сетью, для работы диском, и есть некий модуль (бокс), который отвечает за хранение данных.

Давайте я перейду к докладу. Рассказывать мы будем вдвоем. Александр написал для нас драйвер на PHP. Он будет рассказывать про вещи, связанные с PHP. Я буду рассказывать про разные сценарии, про Lua, про хранимые процедуры на Lua и отвечать на это все с позиции разработчика.

Александр Календарев: Секундочку. Мне просто хотелось бы знать, как много здесь PHP'шников?

(В зале поднимают руки).

Константин Осипов: Порядочно.

Александр Календарев: Да, больше ползала — это уже хорошо. Сегмент рынка PHP — где-то 70%.

hl++ HighLoad++

План

- Архитектура и модель данных NoSQL СУБД Tarantool/Box
- Производительность в сравнении с Redis
- Доступ из PHP
- Data patterns
- Scalability patterns
- Обзор планов

Константин Осипов: Хорошо. О чём доклад. Мы поговорим немного о модели данных. Она у нас тоже своя. Сейчас все изобретают модели данных.

Мы поговорим о производительности. Поговорим о доступе из PHP. Хотя доступ есть из других языков совершенно точно.

Поговорим о pattern'ах. Заранее скажу, что pattern'ов у нас не 12. 12 точно бы не влезли. Но практически все, что вы можете сделать с Redis, вы можете сделать с Tarantool'ом. Тут нужно, конечно, смотреть по использованию памяти. Возможно, где-то одно решение будет выигрывать, где-то другое.

Мы поговорим о том, как делается масштабирование. Пока это не самая сильная наша сторона, но мы можем это делать. Просто этого требует труда.

Также мы поговорим о наших планах.



Я перейду к модели данных. Модель данных Tarantool'a близка, скорее, к реляционной СУБД, но не совсем. Если в реляционной СУБД есть таблицы, у нас это называется пространствами. В пространстве хранятся кортежи.

Кортеж не имеет ограничений по длине. У кортежа может быть 10, 20, 200 полей — сколько угодно. В процессе своей жизни размерность кортежа (dimension) может меняться. Вы можете добавлять, удалять поля. Кортеж — это ассоциация между ключом и списком полей. Ключ — это всегда первое поле кортежа.

Что я на слайде пытаюсь изобразить. По коллекции кортежей, которая из себя представляет space, можно строить индексы. Всегда есть первичный индекс, первичный ключ. Он по первому полу.

Можно строить вторичные ключи. Их можно строить по второму и третьему, 50-му полю и так далее, как вам удобно. Соответственно, если у вас есть индекс по какому-то из полей кортежа, то это поле обязательно должно присутствовать в каждом кортеже, который есть в этом пространстве.

Почему такая модель. В отличие от реляционных систем мы не хотим заставлять вас сразу запланировать вашу схему. Но модель структур данных — это более сложное. Нет изобретения модели как таковой. Это, я бы сказал, *relaxed relational model*. Нет ограничения реляционной модели, но при этом это близко к реляционной.

hl++ HighLoad++

Модель данных (2)

Соответствие терминов:		Типы данных:	Типы индексов:
Tarantool	РСУБД	NUM	HASH
Space	Table	NUM64	TREE
Field	Column	STRING	

Что еще хотелось бы упомянуть. Типы данных, которые у нас есть, это числовой 32-битный, 64-битный и строковый. Вообще о типах данных мы не задумываемся до тех пор, пока нам не нужно строить индексы.

Зачем нужны типы данных Tarantool — просто для того, чтобы правильно их сортировать и хранить в индексах. Если у вас нет индексов, то вы можете в каждом поле хранить какое угодно значение. Мы об этом ничего не знаем. Type agnostic, можно сказать. Вы можете создать кортеж, в котором одни int'ы, потом в тот же самый кортеж записать строки. Если нет индексов, то нет никаких проблем.

Типы индексов — хеши и деревья. Зачем нужны деревья. Деревья позволяют получать доступ к данным в отсортированном виде. Если у вас есть деревья, вы можете получить максимальный элемент, минимальный элемент, проитерировать от одного ключа до +10 ключей, которые больше его в этой коллекции. Дерево — это отсортированная коллекция.

hl++ HighLoad++

Модель данных (3)

Индексы: простые, составные, уникальные, неуникальные
 Операции: INSERT/SELECT/UPDATE/DELETE/REPLACE
 Поддерживается простой SQL

Помимо этого индексы могут быть составными (индекс может покрывать несколько полей, точно так же, как в реляционной БД). Операции, которые мы поддерживаем, это 4 CRUD-операции: INSERT/UPDATE/DELETE/SELECT. Операции выполняются по первичному ключу. Вторичные индексы используются для select'ов. Можно выполнять операцию select по вторичному индексу.

У нас есть SQL, но не для того, чтобы сказать, что мы SQL СУБД. Мы NoSQL или Not only SQL. Просто для того, чтобы упростить. Опять же — мы не изобретаем новый язык программирования для вас. Если вы знакомы с синтаксисом MySQL, то это его подмножество (INSERT/UPDATE/DELETE), пожалуйста, можете использовать просто для доступа к данным.

hl++

HighLoad++

PHP интерфейс

```
$tnt = new Tarantool($host, $port, $admPort);
# $host - хост (def: localhost);
# $port - порт (def: 33013);
# $admPort - административный порт (def: 33015)
```

Александр Календарев: Еще я просто хотел добавить к Косте. Я как пользователь использую SQL из административной консоли, чтобы посмотреть, как у меня легли данные. В основном, для отладки.

Теперь я хочу немного рассказать про наш PHP интерфейс. PHP интерфейс выполнен в виде одного класса. Ничего нового я тут не придумал. У класса есть конструктор, в котором параметры соединения. Это host, port и еще есть так называемый административный порт. Для чего он нужен — там хранится служебная информация. Мы можем делать не только действия с данными, но и посмотреть через определенные команды, какие индексы используются, какие параметры среды и тому подобное.

Теперь о соединении. Соединение здесь сделано отложенным. Все знают, что такое отложенное соединение? Повторять не надо.

hl++

HighLoad++

PHP: INSERT, UPDATE

```
define('SNO', 0);           // номер space
$key = 12345
$tuple = array($key, 'spb', 'Hello Word');
# если данные существуют, они замещаются
$res = $tnt->insert(SNO, $tuple);
$res = $tnt->delete(SNO, $key, [$flag]);
$data = array(1 => 'msk', 2 => 'Hello Bi++!');
$res = $tnt->update(SNO, $key, $data);
# $key - всегда первичный ключ
# $data - асс. массив № поля => нов. значение
```

У нас происходит соединение при первом обращении. Все данные там были по умолчанию (я забыл это упомянуть).

Далее хочу сказать немного о стиле программирования. Так как у нас Tarantool использует только цифры (это сделано для более быстрого доступа), то желательно в какой-то секции объявить все define'ы. У меня define — пространство 0, пространство 1, индекс такой-то. Так проще и понятнее для программиста.

Константин Осипов: Можно я тебя прерву. Что значит — мы используем цифры. Идентификаторы — это на данном этапе цифры. Это что-то вроде объектных идентификаторов. Все пространства пронумерованы, все индексы в пространстве пронумерованы. Нет имен. Если у SQL'ных таблиц можно задать свое имя, у нас просто номер.

Соответственно, протокол бинарный. В этот протокол пакуется номер — опять же для простого доступа, для простого создания всяких прокси и так далее, мультиплексоров именно на протокольном уровне для Tarantool.

Александр Календарев: На слайде представлено 3 оператора изменения данных — INSERT, DELETE и UPDATE. Insert мы выполняем с каким-то пространством. Во всех моих операторах первым параметром идет то пространство, с которым мы работаем. Вторым идут данные.

Данные здесь выполнены в виде массива. Массив представляет собой перечень каких-либо данных. Это может быть как цифры, так и строки (о чем Костя говорил).

Второе — delete. Представляется только первичный ключ. Есть некий флаг. Флаг по умолчанию first. Если это трое, то у нас возвращаются те данные, которые были удалены. Зачем это нужно, мы расскажем далее.

Update. Чем отличается update от insert. Тем, что update может производить изменения конкретных полей. Эти конкретные поля показаны в массиве. Допустим, мы изменяем первое поле. Там 'spb' было, здесь — на Москву ('msk'). Второе поле — было 'Hello Word', изменили на 'Hello Hi++'.

Константин Осипов: Я хотел бы тоже добавить. У всех операций есть возможность возвращать кортеж, с которым они работали. Если вы сделали insert, вы можете сразу получить его назад update. Вы можете получить назад update при delete. Вы сразу можете получить назад кортеж, который вы удалили.

The screenshot shows a web-based interface for the HighLoad++ system. At the top, there's a logo with 'hl++' and the text 'HighLoad++'. Below it, the title 'PHP интерфейс: SELECT' is displayed. Underneath the title is a code block containing a PHP script. The script uses the \$tnt object to perform a select query with parameters: \$count = \$tnt->select(\$SNO, \$index, \$key, [\$limit, \$offset]);. A multi-line comment explains the parameters: # \$key - ключ, возможен массив; # \$index - номер индекса, по которому # осуществляется выборка; # default \$limit = 0xFFFFFFFF, \$offset = 0; # Возвращает количество найденных кортежей # Выборка данных осуществляется методом \$tuple = \$tnt->getTuple();. The entire code block is enclosed in a light gray box.

```
$count = $tnt->select($SNO, $index, $key, [$limit,  
$offset]);  
# $key - ключ, возможен массив;  
# $index - номер индекса, по которому  
# осуществляется выборка;  
# default $limit = 0xFFFFFFFF, $offset = 0;  
# Возвращает количество найденных кортежей  
# Выборка данных осуществляется методом  
$tuple = $tnt->getTuple();
```

Если есть какие-то вопросы — короткие вопросы задавайте сразу, длинные откладывайте до конца, будет время для вопросов. Пожалуйста.

Вопрос из зала: Автоключ есть?

Константин Осипов: Мы об этом расскажем. Вопрос был — есть ли автоинкрементные ключи.

Вопрос из зала: Скажите, пожалуйста, (неразборчиво, без микрофона, 21:08) на PHP — она написана на самом PHP или (неразборчиво, без микрофона, 21:12)?

Александр Календарев: Это extension. На самом PHP это было бы очень медленно. Поэтому не имеет смысла.

Вопрос из зала: Первый ключ можно заменить?

Константин Осипов: Можно ли сделать замену первичного ключа.

Александр Календарев: Можно.

Константин Осипов: Каким образом, я не совсем понимаю.

Александр Календарев: Можно, но не нужно.

Вопрос из зала: Индекс 0 и новое значение.

Константин Осипов: Через одну операцию все-таки нельзя. Вы должны удалить. Update всегда идентифицируется ключом. Вы предлагаете в update'e? Я думаю, что да, можно.

Александр Календарев: Я экспериментировал — можно. Но практически — я не нашел, для чего это нужно.

Константин Осипов: Пожалуйста.

Вопрос из зала: На предыдущем слайде были host и port, куда коннектиться. Если несколько инстансов?

Константин Осипов: Вопрос был: если несколько инстансов, как определять host и port. Это решает Tarantool Proxy, о нем мы поговорим.

Вопрос из зала: Составной ключ. Как будет происходить работа с составным ключом?

Константин Осипов: Вопрос был: как происходит работа с составным ключом. В примере (на слайде) в качестве ключа передается скаляр (ключ — это либо число, либо строка). Но вы также можете передать массив. Это будет составной ключ.

Вопрос из зала: В качестве составного ключа передается array.

Александр Календарев: Да, у меня реализовано. Здесь представлен оператор select, который делает выборку. Также есть оператор multiselect (mselect). Это аналог multiget'a в Memcached. Я о нем чуть позже расскажу.

Что из себя представляет select. Это номер пространства, с которого мы делаем выборку, номер индекса и сам ключ. Номер индекса, как мы говорили, может быть простым, может быть составным. Я просто до этого чуть-чуть не дошел. Если у нас простой индекс, то key у нас либо число, либо строка. Если у нас индекс составной, то key у нас представляет собой массив. При том первый элемент массива — первый индекс, второй элемент массива — второй индекс, следующий массив.

Константин Осипов: Компоненты индекса. Первые и вторые компоненты.

Александр Календарев: Дальше есть два необязательных параметра — limit и offset. Это аналог MySQL — limit и offset.

Константин Осипов: Соответственно, limit и offset имеет значение, только если вы выбираете по неуникальному ключу. Если вы выбираете по уникальному ключу, у вас всегда один кортеж в результате. Либо, если у вас mselect, multiget, то есть вы выбираете кучу ключей сразу, но хотите первые 10 из этого диапазона ключей, которые вы передаете в Tarantool.

Вопрос из зала: Обращение к полям по идентификаторам не является ли опасным в случае изменения структуры данных?

Константин Осипов: Не является ли опасным обращение к полям по идентификаторам в случае изменения структуры данных. Конечно, является. Но это просто вы имеете дело с продуктом, который начинает свой жизненный путь. Мы, безусловно, будем поддерживать имена, но мы будем поддерживать их на уровне синонимов.

Если в базе данных вы всегда привязываетесь к имени столбца, то у нас просто уникальным идентификатором будет число. Мы дадим возможность задавать синонимы для чисел любого рода. Это просто у нас в планах, мы об этом не упомянули. Сейчас физически вы всегда обращаетесь по номеру, да.

Вопрос из зала: Лучше не удалять поля?

Константин Осипов: Если вы удаляете поля... На практике такой проблемы у нас пока не было. Везде, где используется, поля удаляются. Но если они удаляются, то этот кортеж

представляет из себя список, или какую-то такую структуру, или FIFO. Те поля, которые вы удаляете, у которых относительный индекс может измениться, вам это и надо. Нужно, чтобы другое поле съехало на его место по номеру, чтобы вы получили другое поле. На практике такой проблемы пока не было.

Вопрос из зала: Получается, у нас пространство на каждый запрос указывает. Насколько это актуально? Может быть, пространство — это свойство объекта, так как на разных пространствах разные объекты. Какова идеология пространства?

Константин Осипов: Пространство — это коллекция. Вы должны указать, в какую коллекцию вы обращаетесь. Есть поддержка большого количества коллекций. Идеология простая. Пространство — это опять же таблица. Как угодно можете смотреть на это. Я думаю, что в большинстве NoSQL СУБД это называется коллекцией. И в Mongo, и в Couch, и в MemcacheDB. Мы можем двигаться дальше?

Александр Календарев: Сейчас, я еще недорассказал. Я хотел сделать сравнение. Если у нас есть, допустим, запрос по составному ключу, то аналог SQL-оператора будет select*from имя таблицы в key1 and key2. Если это у нас несоставной ключ, то без and.

Также я тут упомянул про оператор mselect, который имеет точно такой же синтаксис. Там ключ представляет собой массив ключей. Выбор тогда будет аналогичен: select*from в key in и множество ключей. Разнообразность select'a.

Второй такой момент, который хотелось бы сказать. Что mselect, что select возвращают количество имеющихся данных. Они не возвращают сами данные. Сами данные возвращаются по циклу оператором getTuple'а.

Константин Осипов: Пожалуйста, ваш вопрос.

Вопрос из зала: У меня был как раз вопрос по поводу select'a.

Константин Осипов: Отлично. Значит, мы ответили на него. Тогда давайте двигаться дальше.

hl⁺⁺ **HighLoad++**

Производительность

Intel I5 , 4G RAM, 7200 RPM SATA
10 потоков, 200-300 байт кортеж
Redis: 120k writes, 270k reads
Tarantool: 100k writes, 260k reads

Я сейчас хотел бы поговорить о производительности. На самом деле, я лично не верю ни одному benchmark'у, который я не написал и не выполнил сам. Поэтому я призываю вас мне не верить. Тем не менее, для нас мы выполняли какие-то benchmark'и, чтобы вообще понять, где мы находимся.

Самый интересный опыт, который мы получили, это что просто DD, запись из Zero в файл на обычной машине дает производительность около 50-60 (ну, до ста) мегабайт в секунду. Большинство NoSQL-решений, включая Tarantool, дает на порядок меньше — 5-10 мегабайт в секунду в записи.

В цифровом отношении это выражается... Эта машинка Intel I5, 4 гигабайта RAM... Машинке где-то года 3. На производительность напрямую влияет даже не скорость диска, а скорость

шины, памяти и процессора. Мы добились 120 тысяч запросов в секунду для записи, 270 тысяч (это для Redis'a). Tarantool — 100 тысяч запросов в секунду по записи и 260 тысяч по чтению.

Сейчас я вам покажу более современную машинку — Intel i7.

(Демонстрация графика, нет в презентации).

Не знаю, насколько виден этот график. Мы здесь говорим о цифрах порядка 700 тысяч запросов на чтение и 300 тысяч запросов на запись. Конкретно в этом сконструированном benchmark'e мы выигрываем у Redis'a. Я покажу, как примерно работает этот цикл benchmark'a.

Вопрос из зала: Что Redis, что Tarantool ?

Константин Осипов: Redis — это синее, Tarantool — это красное. Вы видите, что кривая Tarantool'a находится немного выше. Что здесь происходит. 10 потоков выполняют select'ы в параллель. Это local host benchmark. При сетевом benchmark'e нужно как минимум 200-300 потоков, чтобы выжать эту производительность просто потому, что сетевые задержки скрывают многое. Но 700 тысяч... (Вопрос к залу). У кого здесь есть 700 тысяч запросов в секунду?

Вопрос из зала: Что по оси X?

Константин Осипов: По оси X размер кортежа, который мы выбираем. 200 байт в начале.

Вопрос из зала: Как Redis конфигурировать?

Константин Осипов: Redis конфигурировать с Append Only File, в остальном — настройки по умолчанию. Append Only File включен. Tarantool сконфигурирован, чтобы он синкал данные каждую 0,01 секунды. 100 раз в секунду делается fsync для данных. Размер данных здесь (вы сами можете измерить) — мы вставляем 700 тысяч кортежей. Нет, вставляем мы 300 тысяч. Это график для select'a. Это график для чтения. Вот график для записи.

(Демонстрация графика, нет в презентации).

Вы видите, что запись начинает втупливать на уровне кортежа где-то в 350 байт. Что значит «втупливать»? Мы начинаем около 300 тысяч и заканчиваем на уровне 100 тысяч. При увеличении размера кортежей у нас падает скорость операций.

Вопрос из зала: Это один инстанс?

Константин Осипов: Это один инстанс Tarantool'a, да.

Вопрос из зала: Write — это insert?

Константин Осипов: Write — это insert. Я призываю вас этим benchmark'ам не верить и продемонстрировать нам, что мы недостаточно производительная СУБД. Еще я могу сказать со своей стороны, что мы активно работаем в сторону приближения к dd, но к dd не из dev_zero. Над производительностью мы продолжаем работать.

Вопрос из зала: Скажите, почему такой маленький размер — 300?

Константин Осипов: Что такое 300 байт?

Вопрос из зала: Да.

Константин Осипов: Например, типичный размер сессии, которую вы храните, а пользуетесь из Tarantool, типичное использование — это сессии. Это какие-то данные, которые достаточно ценные, но не настолько, что вы хотите доверить их NoSQL. Это 300 байт.

Вопрос из зала: Например, сортировку я в Redis делаю, достаточно большой объем. Что-то такое есть?

Константин Осипов: Если вы делаете сортировку в Redis'e, вы последовательно добавляете ключи в Redis. Правильно? Потом получаете их в отсортированном виде. Соответственно, в нашем случае вы будете использовать пространство с деревом в качестве индекса. Вы также будете добавлять в это пространство ключи (в нашем случае — кортежи). Потом вы будете получать из этого пространства ваши данные. Если вы делаете сортировку через Redis, то у вас размер запроса равен размеру вашего сортируемого элемента.

Вопрос из зала: Насколько сильно будет тормозить в зависимости от объема сохраненных данных?

Константин Осипов: Классный вопрос. Насколько падает производительность в зависимости от объема данных. Мы in-memory хранилище, поэтому у нас доступ к каждому ключу имеет одинаковую скорость.

В каком смысле мы жжем память. Если у вас сценарий такой, что вы можете кучу данных вытолкнуть на диск и работать только в режиме кеша, у вас только 10% ваших данных используется активно, возможно, Tarantool не для вас. Поэтому производительность остается такой же.

На что влияет объем данных — на время старта и стопа. Если вы стартуете с нуля, у вас куча данных, то вы читаете эти данные с диска. Допустим, 10 гигабайт читается с диска примерно 20-40 секунд с учетом построения индексов и тому подобное. Нужно прочитать все данные в память, построить все индексы. Индексы на диске мы не храним. Мы их строим, когда стараемся.

Вопрос из зала: Если памяти не хватит?

Константин Осипов: Вопрос: если не хватит памяти. Конфигурируется размер памяти, которую вы отводите под кортежи. Соответственно, если памяти не хватает, Tarantool просто не будет позволять оставлять новые данные.

Александр Календарев: Но для этого есть масштабирование.

Константин Осипов: Я еще хотел упомянуть, что типично у вас все-таки в одной ноде не хранится супермного данных. Почему? Типичная конфигурация. Допустим, на машинке 64 гига памяти. Вы там пускаете 2 или 4 ноды, каждой ноде даете по 10 гигов. У вас есть память, чтобы брать снапшоты. Для снапшотов мы используем технику Copy-On-Write.

(Обращаясь к слушателям). Кто слышал о Copy-On-Write? (В зале поднимают руки). Многие слышали. У нас мгновенный консистентный снапшот. Снапшоты обычно администраторы Mail.Ru берут по ночам, когда загрузка минимальная. Если у вас не слишком много update'ов, то ваши снапшоты практически ничего не сожрут.

Вопрос из зала: Я так понял, что вы сравниваете производительность записи просто строковых данных в Redis и Tuple из одного значения в Tarantool?

Константин Осипов: Из двух значений. Ключ значения.

Вопрос из зала: Если говорить о более длинных Tuple'ах, то, наверное, их следует сравнивать в Redis'e со списками.

Константин Осипов: В данном случае вопрос: если мы говорим о таких структурах данных, как списки Redis'a, с чем сравнивать их в Tarantool'e?

Вопрос из зала: Меня интересует производительность.

Константин Осипов: Насколько влияет производительность на размер Tuple. Это классный вопрос. По нашим измерениям, где-то до тысячи элементов вам хватает. Tuple из себя представляет некое упакованное значение в памяти. Если вы постоянно получаете доступ к тысячному элементу, то вам в пределе нужно сделать итерацию по всем.

На диске ваш update хранится в компактном виде. Если вы добавляете 1000-й элемент, 1001-й элемент, то на диск идет запись именно этой команды. «Добавить Tuple такому-то элементу такой-то». Мы не перезаписываем данные на диске всегда, когда вы добавляете или append'ите. Нам нужно просто распаковать и запаковать новый Tuple. По CPU производительность может снижаться при увеличении размера Tuple, но не по диску ввода-вывода.

Я хочу также упомянуть, что одна из наших целей — начать работать с Tuple'ами по 10, 20 тысяч значений. Мы движемся в эту сторону.

Давайте вернемся к слайдам.

The slide has a dark header with the text 'hi ++' and 'HighLoad++'. Below the header, the title 'Auto-increment pattern' is displayed. A table is shown with four rows of data:

1	212-85-01	John Dow
2	212-85-02	Clint Smith
3	967-53-09	Cheryl Wood
4	Новая строка	

Мы даже не дошли до основной части нашего доклада. Я сейчас расскажу про автоинкремент. Я подозреваю, что аудитория должна знать интуитивно, что такое автоинкремент. Автоматическое присваивание первичного ключа новой записи.

The slide has a dark header with the text 'hi ++' and 'HighLoad++'. Below the header, the title 'Auto-increment: PHP' is displayed. A code snippet in PHP is shown:

```

define(S_USER, 1); // номер пространства USER
define(INC_NO, 1); // номер ключа счетчика
define(COUNTER, 1); // номер поля счетчика

$key = $tnt->inc(SNO, INC_NO, COUNTER,
                   [1, true]);
$tnt->insert(NS_USER, $key, $data);

```

Александр расскажет, как это делать в PHP, я расскажу, как это делать с помощью хранимых процедур на Lua.

Александр Календарев: Здесь, в принципе, все программисты. Все интуитивно понятно. Есть операция increment. Это упрощение update. Это увеличение на единицу конкретного поля. Значение пространства, номер пространства, номер ключа, по которому мы делали, и номер поля, по которому увеличиваем. По умолчанию это всегда «единичка». В некоторых случаях нам нужно увеличить на «двоичку», на «троичку». Пожалуйста, это необязательные поля. Также есть значение флага, которое возвращает это поле.

Что мы сделаем. Мы в отдельном пространстве имен (я его называю «служебное», как правило, это нулевое, мне просто так удобно) храним разные служебные счетчики, которые об-

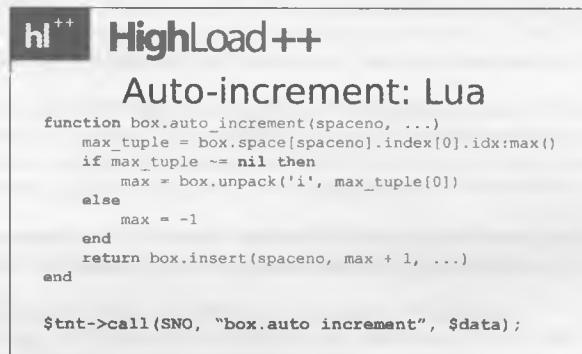
служивают структуру других пространств. Дальше мы увеличиваем значение этого счетчика. Следующей операцией мы просто-напросто вставляем данные.

Так же auto-increment работает и в MySQL'e. Но здесь есть одно узкое место. (*Вопрос к залу*). Кто скажет какое?

Константин Осипов: Здесь, на самом деле, нет узкого места.

Вопрос из зала: Насколько велика вероятность конфликта при развитии в разных потоках, что функция increment выдаст одинаковое значение?

Константин Осипов: Я просто поясню. Здесь два пространства. Каждый update в Tarantool'e атомарный. Соответственно, ваш update возвращает вам значение, которое вы заапдейтили. Вы получаете свое уникальное значение инкремента всегда. Конфликтов в этом конкретном случае быть не может. Но мы сейчас продемонстрируем еще более простой способ, а именно сделать автоинкрементную функцию на Lua.



В Tarantool'e вы можете создавать свои хранимые процедуры. Из хранимых процедур есть доступ к базовым функциям сервера. Хранимые процедуры — это такой клиент, который выполняется внутри сервера. Можно делать insert, update, delete, select. Можно получить Tuple, его распаковать и взять какие-то поля. Можно взять данные и послать их клиенту из хранимой процедуры.

Как работает хранимая процедура. Все, что вы возвращаете из хранимой процедуры, возвращается на клиент. При этом если одна хранимая процедура вызывает другую хранимую процедуру, все, что вы возвращаете из вызываемой хранимой процедуры, просто возвращается как возвращаемое значение функции в любом языке программирования.

Как это можно воспринять. Наверху сидит главный вызывальщик. Все, что ему возвращает хранимые процедуры, он отправляет клиенту в итоге.

На слайде мы видим, что у нас в Lua есть доступ к Tarantool'u как к некоему embedded хранилищу. В модуле box есть массив всех пространств, к которому мы можем обращаться по номеру пространства, в котором есть массив всех индексов, по номеру индекса. Индекс — это объект. У этого объекта мы можем вызвать метод — max, min, текущее число значений в индексе (узнать количество значений). Можем проитерировать по индексу и так далее.

Как здесь работает автоинкремент. Мы просто вызываем хранимую процедуру, которая получает текущий максимум из индекса в этом пространстве и вставляет значение с max+1. Все это выполняется атомарно на сервере. (*Вопрос к залу*). Кому непонятно?

Вопрос из зала: После max там индексы лочим?

Константин Осипов: У нас lock'ов нет. Этот кусочек выполняется атомарно без прерываний. Понимаете, большинство СУБД теряют как раз на lock'ах. За свою работу в MySQL (я пришел в Mail.Ru из MySQL) я занимался тем, что ускорял производительность локинг подсистемы MySQL.

Александр Календарев: Однопоточный сервер.

Вопрос из зала: Что, если процедура зависнет?

Константин Осипов: Что произойдет, если процедура зависнет. Если вы написали глючную процедуру, которая зависнет, вы сами себе злобный буратино. (Смех в зале). Типичная инсталляция Tarantool'a подразумевает, допустим, 5-10 отложенных процедур под конкретный проект. Они отлаживаются, делаются, но при этом любую процедуру можно обновить на лету. Если у вас процедура с ошибкой, вы можете зайти в административную консоль, перекомпилировать ее, дать ей новый definition — и она тут же начинает использоваться во всех ваших клиентах.

Вопрос из зала: Зависшие как?

Константин Осипов: К сожалению, если у вас зависла процедура, вы на административную консоль не зайдете.

Вопрос из зала: Что в этом случае?

Константин Осипов: Килять его — вот и все.

Вопрос из зала: Два вопроса. Что по производительности? Насколько я вижу, здесь также предлагается вызывать из клиента эту процедуру, которая возвратит вам новый инкремент. Почему бы...

Константин Осипов: Нет, она не возвращает новый инкремент. Она делает вставку. Из клиента мы вызываем процедуру с нашими данными. Она все вставляет.

Вопрос из зала: Ok. Тогда что по производительности?

Константин Осипов: Вопрос: что по производительности процедуры. Это, на самом деле, ответ еще на один вопрос, который мне задают примерно так же часто, как вопрос «почему Tarantool называется Tarantool». Это «почему Lua?». Почему выбрали Lua, а не Python, JavaScript — что угодно.Lua — это один из самых быстрых Just-in-time компилируемых языков. Пожалуй, самый быстрый по benchmark'ам. Поэтому он и был выбран. Оверхед этой процедуры вы не заметите. Это Just-in-time compiler. Мы используем LuaJit 2.0 в нашем сервере.

Вопрос из зала: Все-таки, если процедура зависла, что с клиентом будет? Он тоже зависнет?

Константин Осипов: Да, клиент тоже зависнет. Смотрите: здесь нечему виснуть, если у вас все правильно написано. Вы отлаживаете ваши процедуры более или менее... Пока что мы не дошли до того уровня, что...

Вопрос из зала: Хорошо. Если вылетела память или еще какие-то...

Константин Осипов: Что происходит, если в процедуре возникает ошибка. Tarantool внутри себя использует исключения.

Вопрос из зала: Он вернет их?

Константин Осипов: Да, любая ошибка хранимой процедуры возвращается клиенту в виде ошибки err.proc.lua. У нас есть набор стандартных ошибок. Они все задокументированы. До-

пустим, если у вас (неразборчиво, 45:19), мы вам возвращаем (неразборчиво, 45:21). Если у вас исключение в Lua, мы возвращаем исключение в Lua.

Что интересно — Lua может вызвать... Допустим, `box.insert` в конечном итоге вызывает код, написанный на С. Точнее, мы используем Objective C. Если в этом коде произойдет исключение, то исключение пройдет сквозь хранимую процедуру и опять же вернется клиенту. Это одна из причин использовать LuaJit, потому что она совершенно шикарно сквозь себя прорасывает исключения на GCC. Мы используем GCC для компиляции. Обработка ошибок выглядит очень просто.

Вопрос из зала: Можно еще вопрос. Вы сказали, что эта процедура выполняется атомарно. Что это означает? Это означает, что если другой клиент вызовет эту процедуру, то что произойдет?

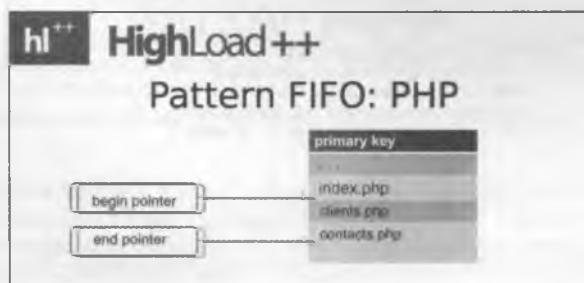
Константин Осипов: Что значит атомарность выполнения процедуры? Это значит, что не происходит переключения контекста, пока процедура выполняется. Если одну и ту же процедуру одновременно вызвало 10 клиентов, то процессор их выполнит одну за другой. Для клиентов это будет выглядеть параллельно. Давайте не будем забывать, что производительность современных процессоров в 100 и 1000 раз быстрее производительности диска и сети.

В типичном случае просто для вас это выглядит параллельным. В этом, собственно, и ключ архитектуры Tarantool'a. За счет современной скорости процессоров мы можем все выполнять последовательно, но для клиентов это будет выглядеть параллельно, потому что все равно они осуществляют доступ по сети или через диск.

Вопрос из зала: Ваша медленная (в ней, допустим, итерация по ста тысячам элементов в пространстве), то да, все остальные будут ждать. Я об этом упомяну.

Константин Осипов: Простите, я уточню. Все остальные этой ноды?

Константин Осипов: Все остальные клиенты этой ноды. Погнали дальше.



Сейчас мы расскажем о pattern'e FIFO. Мы все-таки заявили наш доклад не как Q&A-сессию, а как рассказ о pattern'ах, поэтому будем пытаться.

Что такое FIFO, объяснять не буду. Предположим, у нас есть некий список страниц, которые пользователь посещал. Последние 20 страниц.

hl ++ HighLoad++

Pattern FIFO: PHP

```

FIFO_POP:
define(END_PTR, 5); // номер ключа End Pointer
$key = $tnt->inc(NS, END_PTR, FD_COUNTER, -1,
                   true);
$data = $tnt->delete(NS_USER, $key, true);

FIFO_PUSH:
define(BEG_PTR, 4); // номер ключа Beg Pointer
$key = $tnt->inc(NS, BEG_PTR,
                   FD_COUNTER, 1, true);
$tnt->insert(NS_USER, $data);

```

Вот так это может выглядеть на PHP. Опять же с использованием дополнительного пространства.

Александр Календарев: Здесь, в принципе, все очень просто. Первым действием мы увеличиваем счетчик, вторым мы удаляем данные. Если мы оставляем данные, то увеличиваем счетчик на «единицу», получаем новые данные и вставляем данные. Все очень просто. Следующий pattern будет то же самое — на хранимой процедуре у Кости представлен.

Константин Осипов: Да. Тут был классный вопрос, который будет частью ответа на то, что я сейчас спрошу. Какая проблема с этим подходом? У нас есть два пространства. В одном мы храним указатель на начало и конец очереди для данного идентификатора очереди. Во втором пространстве мы храним саму очередь. Соответственно, push — это изменение двух пространств, pop — изменение двух пространств. Какая проблема с этим подходом?

Вопрос из зала: Гонка.

Константин Осипов: Гонка. Ситуация гонки, совершенно верно.

hl ++ HighLoad++

Pattern FIFO: Lua

```

function fifo_push(name, val)
    fifo = find_or_create_fifo(name)
    top = box.unpack('i', fifo[1])
    bottom = box.unpack('i', fifo[2])
    if top == fifo.max+2 then -- % size
        top = 3
    end
    return box.update(0, name, '=p=p=p', 1, top,
                     2, bottom, top, val)
end

```

На Lua вы можете это сделать атомарно. Я бы хотел пояснить одну вещь. Что значит атомарность на данном этапе в Lua? Вы можете отдать управление в Lua. Я это покажу. Вы можете в вашей Lua-процедуре отдать управление. Потом оно к вам вернется. Опять же — использовать кооперативную многозадачность.

Любое изменение данных, на самом деле, отдает управление. Select не отдает управление. В данном случае, когда у нас FIFO реализовано на Lua, мы тоже должны фактически в какой-то момент сделать два update'a. Ситуация упрощается многократно, потому что update, который мы должны сделать, это создание пустого FIFO. Создание пустого FIFO можно сделать в цикле. Пока не получили для данного FIFO Tuple, создаем его.

Дальше. После того как мы нашли наше FIFO... В данном случае я использую схему данных, когда только одно пространство. Мы в этом пространстве храним: первое поле — это иден-

тификатор FIFO, второе поле — начало FIFO, третье поле — конец FIFO. Дальше уже пошел хвост, само FIFO.

Что я здесь делаю. Я получаю FIFO, получаю нужный мне Tuple, вычисляю новые начало и конец и делаю update этого FIFO. Я выполняю команду update. Это ничем не отличается от update'a из PHP, только это сделано на Lua.

(Показывает на слайде). Это имя пространства. Это имя FIFO. Это закодированное, что-то вроде print for. Мы обновляем, у нас тут 3 присваивания (==). Аргументом присваивания является указатель на скаляр Lua. '`=pointer=pointer=pointer`'. Что мы присваиваем. Первое поле в top, второе поле в bottom и поле top в value. Мы помещаем новое поле в top. Так примерно выглядит это на Lua. Это опять же атомарно, здесь нет никаких raise conditions.

hl++ HighLoad++

Pattern Memcache

- есть возможность создавать свои микро-потоки:
- `box.coro.create()`, `box.coro.yield()`
- создаете хранимую процедуру, запускаете её
- получаете *custom expire process*

- используется для хранения сессий в mail.ru:
4 машины, по 2 Tarantool/Box на каждой, 2 мастера,
и 2 реплики
40-60k requests/second, CPU usage < 20%

Еще один важный аспект, о котором я бы хотел поговорить и который отвечает на многие вопросы по поводу того, что произойдет, если процедура зависает.

В Lua есть функции по созданию кооперативных задач. Вы можете создать свою хранимую процедуру, которая будет выполняться как зеленый поток внутри Tarantool'a. Это что-то вроде temporal triggers, повторяющихся событий — как угодно.

Вы можете создать внутри хранимой процедуры... Когда, допустим, выполняются сложные вычисления, вы можете отдать управление. Есть функция `yield`, которая отдает управление другим потокам. Соответственно, вы должны понимать, что если вы отдаете управление, когда оно к вам вернется, данные могут измениться. Другие update'ы, другие процедуры могут работать.

Как мы это используем в Mail.Ru. В Mail.Ru есть одно из центральных хранилищ, где мы храним все сессии всех пользователей. В Mail.Ru я даже не знаю, сколько активных, боюсь сорвать. Те данные, с которыми я работал. 40 миллионов активных пользователей. Общее количество пользователей, допустим, около 100 миллионов. Соответственно, нужно хранить 40 миллионов сессий. Сессия представляет из себя набор «ключ — значение». Для каждого пользователя идентифицируется user id. Речь идет где-то о 80-ти гигабайтах данных.

Сессии Mail.Ru хранят в Tarantool'e. Когда пользователь заходит, сессия создается. Для этого используются 4 машины. 2 машины — master'a, 2 машины реплики. В Tarantool'e есть репликация. На каждом master'e работает по два Tarantool'a.

Как они работают. В первую очередь, это ассоциативный массив ключ — значение. Пользователь зашел — мы создали сессию, пользователь сделал sign out — мы удалили сессию. Кроме того, работает expire process (это как в Memcached), который удаляет сессии по неактивности. Expire process постоянно итерирует по пространству сессий. Это опять же зеленый поток. Он выполняется в параллель с потоками, которые обслуживают клиентов и удаляет старые сессии.

Но что интересно — чего вы не можете добиться в Memcached или в любом другом сервере, который поддерживает expire. Когда мы в Tarantool'е делаем expire, мы не просто удаляем кортеж. Мы выполняем некую custom бизнес-логику при удалении.

В Mail.Ru есть политика, которая подразумевает, что пользователи, которые были неактивны два года, должны быть удалены из системы. Если вы не пользуетесь почтой вообще два года, вас удаляют. Когда мы делаем expire-сессии, мы удаляем большую кортеж-сессию в 500 байт из пространства сессий и кладем маленький кортеж user id data в пространство «кладбища». В «кладбище» мы храним эти сессии, которые мы разлогинили.

Есть отдельный процесс, который периодически проходит по «кладбищу» и видит, что если на этом «кладбище» юзер лежит уже два года (он не заходил два года), мы этого юзера можем удалить. В чем тут идея. Вы можете с использованием хранимых процедур закодировать свою custom логику, например, по expire.

Я упомянул, что CPU — наш главный ограничитель — меньше 20% на этой системе используется.

Вопрос из зала: Как сделать шардинг на эти 4 Tarantool'а?

Константин Осипов: Вопрос: как сделать шардинг на 4 Tarantool'а. Просто хеш от user id.

rizaevka@mail.ru	2011-05-09	SPB
slavsysper@mail.ru	2011-09-09	MSK
oxic10@mail.ru	2011-10-01	SPB

Это просто картинка, которая демонстрирует, что у нас есть вторичные ключи. Мы сейчас будем рассказывать про Tarantool Proxy.



Это application, который написал Александр на PHP как раз для шардинга.

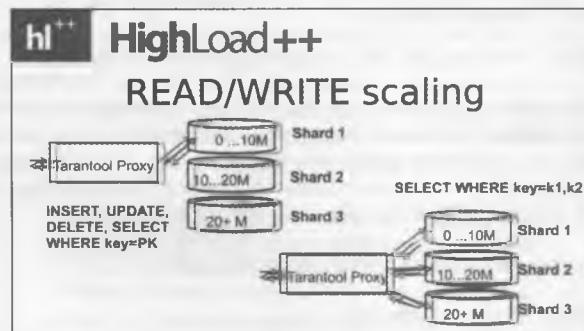
Александр Календарев: Он не на PHP написан.

Константин Осипов: На С.

Александр Календарев: Да. Из чего у нас шардинг состоит. Есть некий алгоритм, по которому мы шардим данные. Алгоритм может быть забит либо в config'и. Допустим, с первого по 10-миллионный user id у нас хранится на сервере 1 с такими-то параметрами коннекции. Следующие 10 миллионов хранятся на сервере 2 и так далее. Либо это может быть какая-то функция. Есть возможность написать свою custom'скую функцию.



Что из себя представляет. Это многопоточный демон. Он проксирует запросы, определяет номер, куда проксировать, и отправляет ответ обратно.



В Tarantool'е существует 2 вида операций. Это операции по первичному ключу и операции по вторичному ключу. Операции по первичному ключу приходят на Tarantool Proxy, по config'у или по custom'ской функции определяется номер шарда и отправляется на соответствующий сервер.

Если у нас существует операция по вторичному ключу (а по вторичному ключу у нас только один select), то мы не знаем, на каком шарде все эти данные. Поэтому надо проверить все шарды. Просто идут запросы на все сервера, возвращаются. Дальше эти данные компонуются и отправляются обратно клиенту.

В принципе, здесь все понятно. Я готов ответить на вопросы. Если я буду рассказывать больше, уже времени нет.

Вопросы и Ответы

Вопрос из зала: Получается, что Proxy должен быть один, потому что мы lock'и получим. Правильно?

Александр Календарев: Proxy может быть несколько.

Вопрос из зала: Как мы тогда lock'ов не получим? При insert'e.

Константин Осипов: Позвольте пояснить. У вас insert всегда выполняется по первичному ключу. Соответственно, задача Proxy — просто определить шард, на который его послать. Дальше на этом шарде все выполняется атомарно.

Вопрос из зала: Этот широковещательный запрос на шарды отправляет запрос параллельно или последовательно?

Александр Календарев: В настоящее время он направляет запросы последовательно. Над тем, чтобы он параллельно отправлял запросы, я буду работать. Просто сперва надо довести до ума этот, потом будем улучшать. Над проектом я работаю всего 4 месяца. Не успею все.

Вопрос из зала: Как сделать балансировку в данном случае? Что произойдет?

Александр Календарев: В данном случае балансировка не предусмотрена. Но балансировка заложена в проекте. Как только я сделаю параллельные запросы, я сразу займусь балансировкой. Здесь мы отправляем только на один сервер, а балансировка предполагает, что у нас, допустим, есть не одна, а две шарды.

Допустим, шарда 1A находится на сервере А, шарда 1B находится на сервере В. Из сервера А идет репликация на сервер В. Если у нас вылетает сервер А, то балансировщик переключит на сервер В. Вот что подразумевает балансировка. Или robin rabin сделаем.

Вопрос из зала: Я имел в виду, что логичнее данные раскидывать параллельно на несколько серверов. Вопрос — как это сделать?

Александр Календарев: Репликация. Мы кидаем данные на сервер А...

Константин Осипов: Александр, позволь я отвечу. Вы с помощью плагина можете выбрать алгоритм определения шарда. Соответственно, вы можете просто, как здесь сделано, выбирать шард по диапазону, а можете выбирать шард по хешу. У вас будет равномерное распределение по ключу. Но это опять же не решает проблем супергорячих ключей и так далее. Давайте вы подойдете к нам в конце, потому что нас уже выгоняют.

Вопрос из зала: Скажите, как используется авторизация?

Александр Календарев: Авторизации нет.

Константин Осипов: Tarantool не предназначен для выставления на всеобщее обозрение. Это что-то, что работает в интернет, поэтому авторизации у нас на данном этапе нет.

Я призываю всех подойти к нам с вопросами. Если вы хотите попробовать, обязательно возьмите у меня контакты. На все вопросы лучше отвечать еще в Skype. На лету я окажу всю помощь.

Большое спасибо!

Управляемый code injection: как мы считаем все пользовательские отчеты за один проход в системе интернет-статистики Openstat

Михаил Якшин



Михаил Якшин: Здравствуйте! Добрый день всем! Мы начинаем эту секцию докладом от Openstat об управляемом code injection'e в Apache Hadoop, или как мы считаем все пользовательские отчеты за один проход в системе интернет-статистики Openstat.

О чём этот доклад

- Какие бывают отчёты в веб-аналитике?
- Как эти отчёты реализуются традиционно и не очень традиционно?
- Как посчитать много отчётов в одном потоке?
- Какую пользу из этого можно извлечь?

О чём этот доклад.

Я расскажу о том, какие вообще бывают отчеты в веб-аналитике. Какие запросы мы получаем от клиентов. Какие вообще бывают отчеты — стандартные, нестандартные. Как эти отчеты реализуются традиционно и не очень традиционно.

Как можно посчитать много отчетов разнородных в одном потоке так, чтобы сэкономить время и вычислительные ресурсы кластера. Как оценить пользу, которую мы из этого извлекаем.

Веб-аналитика: взгляд с высоты птичьего полета



Если посмотреть на веб-аналитику с высоты птичьего полета, то все очень банально и просто.

Есть сайт, на котором стоит счетчик. Сайт генерирует логи веб-сервера. Счетчик собирает информацию, генерирует логи счетчика. И те, и другие как-либо обрабатываются, и получаются отчеты.

Типы отчётов

- **Стандартные отчёты** – предоставляются всем клиентам по умолчанию, не требуют никакой дополнительной настройки
 - Хорошо параллелизуются и считаются на потоке
- **Нестандартные отчёты** – требуют понимания специфики задач клиента, дополнительной настройки или программирования
 - Возникает проблема с параллелизацией и масштабированием

Если посмотреть на отчеты, то можно заметить, что их можно разделить на два типа: стандартные и нестандартные.

Стандартные предоставляются всем клиентам системы веб-аналитики по умолчанию, не требуют никакой дополнительной настройки, знания специфики клиента. За счет того, что клиентов много (их, как правило, порядка миллиона), они хорошо параллелизуются и считаются на потоки.

Нестандартные отчеты требуют понимания специфики задач клиента, знания о том, что ему надо, каких-то сложных параметров, которые нужно посчитать. Требуют, как правило, дополнительной настройки какого-то программирования. Так как они нужны всего-то одному-двум клиентам, тут же возникает проблема с параллелизацией и масштабированием.

Если их запускать так, как они запускаются стандартно, то при большом количестве запросов на такие отчеты, все будет очень медленно и долго.

Типы отчётов

Стандартные:

- Популярные страницы
- Точки входа, точки выхода
- Источники трафика
- Браузеры
- Операционные системы
- Экранные разрешения
- Мобильные устройства
- География

Нестандартные:

- Разделы сайта
- Характеристики посетителей
- Популярные темы и комментарии в форуме
- Продажи интернет-магазина
- Каналы привлечения аудитории
- SEO

Стандартные и нестандартные отчеты представлены на этом слайде. Я сейчас зачитывать не буду. То, что в колонке слева (стандартные), это более или менее известные всем отчеты, доступные практически в любой системе веб-аналитики. Можно открыть Google Analytics, Yahoo, Omniche. У нас они точно так же есть.

Нестандартные отчеты все характеризуются тем, что нужно знать какую-то специфику сайта клиента и делать какие-то дополнительные настройки для этих отчетов. Например, разделы

сайта. Требуют передачи информации о том, что именно составляет этот раздел, какой-то функции mapping'a URL в названии раздела. Или передачи вместе с регистрацией событий на счетчике или в логах какой-то информации о том, какой это был раздел из приложения.

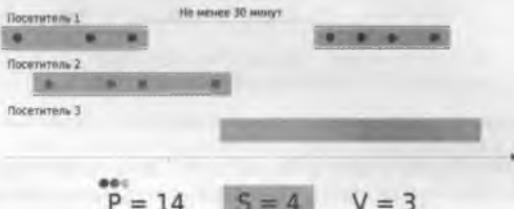
Тридцатисекундный экскурс в термины веб-аналитики

- **Просмотры (Pageviews)** – события загрузки страницы сайта посетителем
- **Визиты/сессии (Visits/Sessions)** – последовательность обращений посетителя к сайту, интервал между которыми не превышает 30 минут
- **Посетители (Visitors)** – пользователи, совершившие обращение к сайту и идентифицируемые некоторым образом (по IP, cookie и т.д.)

Очень быстрый, 30-секундный экскурс в термины веб-аналитики. Мы будем говорить о просмотрах (Pageviews). Это банально события загрузки страницы сайта пользователем, посетителем. Эти просмотры и некие другие события, кроме просмотров, группируются в сессии/визиты. Это последовательность таких обращений от одного пользователя, интервал между которыми не превышает 30-ти минут.

Также есть сами посетители (Visitors) — пользователи, которые совершили обращение к сайту и которых мы можем как-то идентифицировать. Это важно. Например, по IP, по cookie, по комбинации этого всего с юзер-агентом, с настройками proxy. Есть еще какое-то количество уникальных данных, по которым можно ухватиться.

Про термины в картинках



Если показать это в картинках, выглядит это ровно так. Точки — это Pageviews. Прямоугольники, которыми обведены точки, это сессии, или визиты. Кстати, такое неудачное название термина. Происходит много clash'ей между понятиями visit и visitor, хотя это совершенно разные вещи.

Посетители, которые расположены по вертикали, это разные посетители. Таким образом, получается, что на картинке у нас представлено 14 Pageviews, 4 сессии и 3 посетителя.

Какие отчёты заказывают потребители веб-аналитики? (1)

Новости	4905	490	301
Пресс-релизы	3691	527	301
Магазин	4548	413	391
Техподдержка	2548	509	364
Файлы	4092	511	378

Какие отчеты, если говорить о нестандартных отчетах, заказывают потребители веб-аналитики. Типичный совершенно отчет (самый простой, который можно придумать) — это отчет по разделу. Либо из названия страницы (из URL'a или из Title'a), либо с помощью передачи какой-то переменной происходит генерация этого самого token'a, который попадает в левую колонку отчета «раздел». Происходит группировка всего трафика, который попал на счетчик или в логах сервера, по этому самому разделу.

Мы рассчитываем 3 основных показателя — число Pageviews, число сессий, число посетителей. Соответственно, получается такой одномерный отчет, как мы его называем. Одномерный, потому что измерение здесь, по сути, всего одно — раздел.

Какие отчёты заказывают потребители веб-аналитики? (2)

Пол → Раздел ↓	Мужской	Женский	Всего
Новости	80	15	95
Статьи	60	50	110
Магазин	50	100	150

На пересечении — один из показателей, например, P — число pageviews

Двумерный отчет. Здесь можно заметить уже две оси. Одна идет вниз — «разделы». Направо идет ось со значениями: пол «мужской», «женский» и «всего». На пересечении мы, таким образом, показываем только один из показателей, который выбран. Получается такой двумерный отчет.

Какие отчёты заказывают потребители веб-аналитики? (3)

Группа ↓	Источник → Товар ↓	SEO	Поисковики	Контекст	Прямые переходы
Кабель	UTP	30	90	60	100
	STP	40	20	0	740
Коммутаторы	RJ45	80	150	300	30

На пересечении — один из показателей, например, V — число уникальных посетителей

Трехмерный отчет уже представить сложнее, но выглядит он обычно как-то так. Его, естественно, кубиком никто не рисует. Показывают его планарную версию. В его двух колонках,

например, «группа» и «товар», соответственно, товарная группа и название товара. По горизонтали идет направо источник того самого визита, который привел посетителей к этому товару.

Level 1: сегментация

«Сегменты» – механизм для построения статистики по характеристикам, которые:

- имеют относительно немного интересующих значений, например:
 - пол: мужской / женский
 - география: Москва / остальная Россия / зарубежье
 - совершил ли посетитель целевое действие: да / нет
- не изменяются на протяжении визита

Каким же образом можно дать возможность считать такие отчеты, и какие вообще есть подходы к расчету таких значений в отчетах.

Самый простой способ, который более или менее везде есть (много где есть), известен и – самое главное – всем понятен. Это механизм сегментов. «Сегменты» – это механизм построения статистики по характеристикам, которые имеют относительно немного интересующих значений, так что можно создать сегмент на каждое из этих значений.

Например, пол – мужской/женский. География – для какого-то конкретно выбранного региона. Или ответ на вопрос «совершил ли посетитель целевое действие»: да или нет.

Значения сегмента не изменяются на протяжении визита. Здесь я немного вру. На самом деле, изменяется, но максимум один раз. Оно может переключаться с нолика на единичку и никогда не переключается обратно.

Пользовательские сегменты

- Сегмент – бинарная характеристика визита; визит либо относится к сегменту (1), либо не относится к нему (0)
- Сегмент задается в виде булевой функции, оперирующей полями лога, например:
 - se_name="Google"
 - geo_country="RU" AND browser="Firefox"
- Если хотя бы в одном событии визита функция вернула true, то весь визит относится к сегменту

Таким образом, сегмент – это бинарная характеристика визита. Визит либо относится к сегменту (1), либо не относится (0).

Сегмент задается в виде некоей булевой функции, оперирующей на переменных со значениями из полей лога. Например, поисковая система Google. Или: страна Россия, браузер Firefox (`geo_country=>RU` AND `browser=>Firefox`).

Если хотя бы в одном из событий визита функция вернула true, то весь визит относится к сегменту.

Иллюстрация работы сегмента

Условие: `se_name = "Google"`
 ("посетители, пришедшие на сайт с Google")



Проиллюстрировать можно следующим образом. Берем такое условие, например. Почему такое — потому что, как правило, в визите pageview с таким условием будет всего один. Пользователь приходит на сайт всего один раз. Вот эта переменная получается ровно из реферера, из которого пришел пользователь, который будет в его хите.

Иллюстрация работы сегмента

Условие: `se_name = "Google"`
 ("посетители, пришедшие на сайт с Google")



Например, у нас есть такие значения этого `se_name`. В третьем хите одно, в первом хите другой сессии — другое. Таким образом, проходя этим алгоритмом расчета bitmask'ов сегментов, мы сначала определяем, что третий хит подпадает под сегмент.

Иллюстрация работы сегмента

Условие: `se_name = "Google"`
 ("посетители, пришедшие на сайт с Google")



Этот хит у нас переводит все на весь сегмент целиком.

Иллюстрация работы сегмента

Условие: `se_name = "Google"`
 ("посетители, пришедшие на сайт с Google")



Тем самым все события этой самой сессии, подпавшей под этот сегмент, становятся тоже удовлетворяющими условиям сегмента.

Как это выглядит в веб-интерфейсе			
Показатели			
	Номер	Показатель	Значение
Посещаемость	1234567	Всего	51352
Посещаемость	1234567	0	5336
Посещаемость	1234567	1	33
Посещаемость	1234567	Ростелеком	31
Посещаемость	1234567	Корбина	2270
Посещаемость	1234567	Билайн	1637
Посещаемость	1234567	0	—
Посещаемость	1234567	1	—
Посещаемость	1234567	Всего	1584
Посещаемость	1234567	0	31
Посещаемость	1234567	1	33
Посещаемость	1234567	Ростелеком	1637
Посещаемость	1234567	Корбина	104
Посещаемость	1234567	Билайн	104
Посещаемость	1234567	0	—
Посещаемость	1234567	1	—

В веб-интерфейсе это выглядит примерно следующим образом. Любой отчет — стандартный, нестандартный... Как правило, проще всего добавить сегмент к какому-то стандартному отчету вплоть до отчета «посещаемость», который, по сути, есть просто расчет посещаемости в тех или иных показателях по всему сайту, по всему трафику в целом.

Здесь я попробовал проиллюстрировать расчет провайдеров. В левой колонке — вся аудитория, это так называемый нулевой сегмент (дефолтный). Справа мы выделили сегмент «мобильные пользователи». Пользователи с мобильных устройств. Мобильность устройства определяется по user агенту. User агент в течение сессии не меняется. Все нормально, можно использовать сегмент. Такой хрестоматийный пример.

Как это выглядит внутри в отчёте			
ID счётчика	Сегмент	Провайдер	S
1234567	0	Всего	51352
1234567	0	Ростелеком	5336
1234567	0	Корбина	2270
1234567	0	Билайн	1637
1234567	0	—	—
1234567	1	Всего	1584
1234567	1	Ростелеком	31
1234567	1	Корбина	33
1234567	1	Билайн	104
1234567	1	—	—

Внутри в кластере этот отчет выглядит следующим образом. Есть некий идентификатор счетчика. Все строчки задублированы два раза — для нулевого сегмента и для первого сегмента. Тем самым мы не меняем схему данных. Схема данных, несмотря на то сколько бы сегментов ни было, всегда остается такая планарная. Сегменты можно добавлять до бесконечности, до каких-то разумных пределов, пока нам не жалко хранить этот bitmask.



Общая схема работы с сегментами выглядит следующим образом. Логи сначала проходят этап вычисления bitmask'a сегментов. Формируется некий дополнительный раздел данных под названием «битмаски сегментов». Они, join'ясь с логами, поступают в расчет стандартных отчетов. Важно то, что стандартных. Сейчас мы говорим только о стандартных отчетах, которые расширяются этими дополнительными колонками, которые образуются из сегментов.

Получаются некие отчеты, которые уже чуть больше, чем стандартные отчеты. Целый огромный класс задач можно с помощью сегментов решать, таким образом.

Например, решаются хорошо задачи, доходит или не доходит пользователь до совершения какого-то действия, до целевой страницы. Например, хорошо решаются задачи сегментации аудитории на какие-то сегменты — по географии, по технometрике используемых юзер-агентов, по каким-то передаваемым переменным, значения которых не изменяются (например, по полу, по возрасту, по каким-то еще данным из анкеты в социальной сети, например).

Level 2: пользовательские отчёты

Сегменты не годятся в том случае, если:

- Нужно посчитать статистику по характеристике, изменяющейся в течении сессии
 - например, посещаемый раздел сайта
- Число интересных значений характеристики исчисляется сотнями, тысячами, миллионами
 - например, ассортимент товаров в интернет-магазине

Если нам этот механизм становится мал, и мы из него выросли, то мы переходим на уровень 2. Это полноценные пользовательские отчеты. Сегменты в этом случае не годятся.

Если нам нужно посчитать статистику по какой-то характеристике, которая изменяется в течение сессии (хрестоматийный пример здесь — это раздел сайта). Если мы зададим в сегменте условие «страница=URL этого раздела сайта», то мы получим совсем не то, что некоторые думают, что они получают. Мы получим всех посетителей и все сессии тех посетителей, которые заходили в этот раздел сайта. Мы не получим посещаемость именно этого раздела сайта в pageview.

Как было проиллюстрировано ранее, сегмент расползется на всю сессию целиком и даже на те pageviews, которые были сделаны вне этого раздела. Соответственно, появятся единички в bitmask'ах, и они будут засчитаны, видимо, не туда, куда надо.

Или другой сильный побудительный фактор для того, чтобы не использовать сегменты. Когда нам нужна не бинарная характеристика, а тогда, когда число этого домена исчисляется боль-

шим числом, и создавать по сегменту на каждое из этих значений либо сложно, либо вообще невозможно. Время, ассортимент товаров в каком-нибудь большом магазине, постоянные изменяющиеся штуковины, типа трендов на новостных сайтах, еще что-то такое.

Традиционный отчёт на базе Apache Hadoop

- Создается код для нескольких job'ов, реализующих цепочку:
mapper → reducer [→ mapper → reducer → ...]
- На вход подаются все логи
- Нужно следить за корректным запуском job'ов в нужном порядке и передаче промежуточных результатов от одного job'a к другому

Как делаются традиционные такие отчеты на базе Apache Hadoop. Создается код из нескольких job'ов, которые реализуют цепочку mapper → reducer [→ mapper → reducer → ...]. На вход подаются все логи, которые у нас есть, на выходе получаем отчеты.

Нужно следить за корректным порядком и work for запуска job'ов. Чтобы они правильно запускались, правильно обрабатывались их падения, перезапуски. Чтобы мы получали промежуточный результат. Чтобы временные данные, созданные первым job'ом, корректно попадали ко второму и так далее по цепочке.

Все это достаточно затратно и, может быть, неприятно или, как минимум, медленно.

Level up: Hadoop + Cascading

Переходим от терминов map-reduce к использованию готовых примитивов:

- **Function** – преобразование 1 строчки в 0..N
- **Filter** – проверяет условие, оставляет или нет 1 строчку
- **Aggregator** – реализует итератор над группой: initializer, iterator, finalizer
- **Buffer** – предоставляет Java Iterator по группе

Поэтому мы уже довольно давно используем не чистый Hadoop, а Hadoop с продуктом под названием Cascading. Он позволяет нам перейти от терминов map-reduce к использованию более общих и более понятных для особенно непрофильных программистов map-reduce вещей. Таких, как:

- Function — преобразование одной строчки в от одной до многих строчек.
- Filter — фильтрация, преобразование одной строчки в 0 или 1 строчку.
- Aggregator — это итератор над группой, у которого есть iterator, есть initializer (работа над группой) и finalizer. Мы получаем группу, сначала вызываем какую-то функцию, которая начинает обработку группы, потом много раз вызываем для обработки каждого элемента группы функцию и финализируем группу, закрывая ее.
- Buffer — замечательный объект, который предоставляет Java Iterator по группе. По сути, то же самое, что Aggregator, но иногда в более удобном или, наоборот, неудобном для задачи виде.

Cascading: пример

```
// Вычисляем названия разделов из URL
pipe = new Each(pipe, new DeriveSections(), Fields.ALL);
// Задаем группировку
pipe = new GroupBy(
    pipe,
    new Fields(F_COUNTER_ID, F_SECTION_NAME),
    new Fields(F_VIS_ID, F_SESSION_ID, F_ID)
);
// Делаем агрегацию того, что нагрупировали
pipe = new Every(
    pipe,
    new BuildVPSE()
);
```

Извиняюсь за то, что показываю код. Но это относительно важно. Я к нему еще немного вернусь.

В самом простом виде отчет на Cascading выглядит примерно так. Мы вычисляем какую-то функцию от URL, чтобы получить название раздела, например. Задаем группировку и делаем агрегацию того, что нагрупировали.

Насколько сложно написать такой отчёт?

- Основной код: 3 строчки (на предыдущем слайде)
- DeriveSections: ~25 строчек
- BuildVPSE: ~200 строчек
- Трудозатраты: порядка часа-два (50-70% времени - написание тестов)
- BuildVPSE – единожды написанный агрегатор, применяется во всех отчётах

Таким образом, основной код отчета у нас — 3 строчки, условно (если все это записать в строчки). Функция, допустим, 25 строчек, 30 строчек (в зависимости от того, насколько она сложная).

Агрегатор BuildVPSE побольше, но он пишется один раз. Что важно, он, единожды написанный, применяется во всех отчетах. Все эти показатели, как правило, нужны везде и такие.

Трудозатраты на написания такого отчета — порядка часа-двух, большую часть из которых занимает написание тестов.

Чем плох такой отчёт?

- Писать отдельную задачу для каждого отчёта затратно
- Запуск отдельных задач на каждый отчёт - затратен
- Поднимать с диска все данные всех счётчиков для обработки одного отчёта - затратно

Тем не менее, чем плох такой отчет. Писать отдельную задачу для каждого отчета — это затратно. Почему — потому что поток задач большой. Час-два на решение какой-то совершенной типовой задачи тратить грустно.

Запускать отдельные задачи на каждый отчет тем более затратно. Когда логи хранятся в самой общей куче, на вход отчета попадают все логи сразу. Несмотря на то что результат нужен только для какого-то одного счетчика, который составляет 0,001% трафика, нам нужно поднимать с диска, обрабатывать все-все-все.

Как можно улучшить ситуацию?

- Очевидно – найти у отчетов что-то общее и попробовать объединить расчёт всех отчетов в один проход
- На первый взгляд – это кажется сложным

Как же можно улучшить такую ситуацию. Очевидно, попробовать вынести эти возможные действия за скобки и попытаться найти у этих отчетов что-то общее, а внутри оставить ту самую соль, которая составляет отчет.

На первый взгляд, все продемонстрированные отчеты кажутся различными. На самом деле, это не совсем так.

Плоское представление двумерных отчетов

Пол →	Мужской	Женский	Всего
Раздел ↓			
Новости	80	15	95
Статьи	60	50	110
Магазин	50	100	150

Раздел	Пол	Число
Новости	Мужской	80
Статьи	Мужской	60
Магазин	Мужской	50
Новости	Женский	15
Статьи	Женский	50
Магазин	Женский	100
Новости	Всего	95
Статьи	Всего	110
Магазин	Всего	150

Первое, что предлагается в данном случае. По сути, сделать денормализацию. Спуститься от этого красивого пользовательского представления отчетов в многомерном виде к представлению отчетов в более традиционном виде со стабильной схемой. У нас есть фиксированное количество колонок, которое зависит только от размерности отчета, но никак не от того, сколько в этом отчете данных. Если появляется пол «мужской», «женский», «всего» и еще что-то, то расширения схемы в данном случае не происходит.

Здесь продемонстрировано, как это сделать для двухмерного отчета.

Плоское представление трёхмерных отчетов

Группа →	Источник = Типоряд			Код	Контекст	Число
	UTP	STP	RJ45			
Кабель	30	90	60	Кабель	SEO	30
Кабель	40	20	0	Коммутатор	SEO	40
Кабель	80	150	300	Кабель	Поисковики	80

Группа →	Источник = Типоряд	Код	Контекст	Число
Кабель	UTP	SEO	Поисковики	30
Кабель	STP	Поисковики	40	
Коммутатор	UTP	Поисковики	80	
Кабель	RJ45	Поисковики	90	
Кабель	UTP	Контекст	20	
Коммутатор	RJ45	Поисковики	150	
Кабель	STP	Контекст	60	
Кабель	RJ45	Контекст	0	
Коммутатор	RJ45	Контекст	300	

Для трехмерного отчета это тоже делается, как ни странно. Думаю, все поверят, что это делается и для четырех-, пятимерных отчетов и так далее.

Основная идея

Все подобные отчеты можно уложить в одну общую схему:

- Некая специфичная для отчета функция-обработчик
- Группировка
- Агрегация

В чем основная идея. Она частично была уже проиллюстрирована на слайде с кодом. Все подобные отчеты можно уложить в общую схему.

1. Некая специфичная для отчета функция-обработчик.
2. Группировка.
3. Агрегация.

Схематично, в терминах Cascading

```
// Вычисляем пользовательскую функцию
pipe = new Each(pipe, new UserFunction(), Fields.ALL);
// Задаем группировку
pipe = new GroupBy(
    pipe,
    new Fields(F_COUNTER_ID, GROUP_BY_FIELDS),
    new Fields(F_VIS_ID, F_SESSION_ID, F_ID)
);
// Делаем агрегацию того, что нагрупировали
pipe = new Every(
    pipe,
    new BuildVPSE()
);
```

В этом коде нужно
записать только эти
2 вещи!

Если посмотреть на код, то для того, чтобы считать все отчеты, указанные выше, единственное, что в этом коде надо изменять, это две вещи — пользовательскую функцию и поля для группировки. Начнем со второго. Это проще.

Маленькая проблема №1: разные размерности набора полей группировки

- Для одномерного отчета:
 - (раздел)
- Для двумерного:
 - (раздел, пол)
- Для трёхмерного:
 - (группа товаров, товар, источник трафика)

В чем проблема с полями для группировки. В том, что разные размерности полей. На первый взгляд, непонятно, как считать отчеты в одном потоке, у которых одна, две, три, десять колонок. Как это все соотносить. Ответ очень простой.

Решение проблемы разных размерностей: вложенные Tuple

Конвертируем исходный поток, собирая и все поля из условия группировки в одно поле с вложенным Tuple:

- | | |
|--|--|
| <ul style="list-style-type: none">• Было:
[счётчик, раздел, ...]
[счётчик, раздел, пол, ...]• Стало:
[счётчик, [раздел], ...]
[счётчик, [раздел, пол], ...] | Единое поле, которому можно присвоить название и задать это название в условии группировки |
|--|--|

Обернуть эти самые колонки в еще один Tuple, встроенный в тот Tuple, в котором представляются строки отчета, строки логов. Считать этот объект отдельным объектом и группировать по нему. Тем самым, у нас получается стабильное количество колонок вне зависимости от размерности отчета. В любом инструментарии эта группировка будет работать, агрегация будет производиться.

Проблема №2: code injection

Нужно дать возможность выполнять на кластере в рамках первоначального преобразования строчки (Function в терминах Cascading) произвольный пользовательский код:

```
// Вычисляем пользовательскую функцию
pipe = new Each(pipe, new UserFunction(), Fields.ALL);
```

Проблема № 2 чуть посложнее. Code injection. Нам надо предоставить пользователю выполнять в рамках задач на нашем кластере какой-то свой собственный код. У этого есть целый набор плюсов и минусов.

Чем это плохо?

- Функция может быть тяжелой и затратной
- Разумеется, проблемы с безопасностью
- Плохая локальность и много пересчёта: отчёты считаются, как правило, за большие периоды, а результат функции зависит только от конкретной строчки и может быть расчитан прямо при получении строчки

Чем же это плохо. Тем, что, естественно, если пользователь будет писать такую штуку (даже если у него нет каких-то злых намерений), то он банально может написать функцию, которая будет тяжелой и затратной.

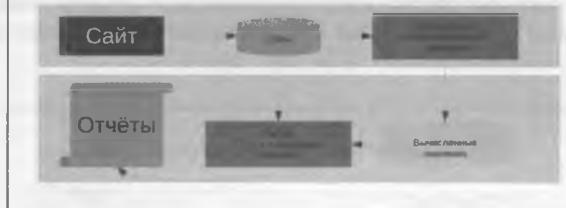
В качестве совершенно банального примера. Были случаи, когда нам писали функции, состоящие из 40-50 килобайт строчек вида «if что-нибудь equals что-нибудь», «что-нибудь equals что-нибудь». Разумеется, при выполнении этой функции в контексте обработки, например, трафика порядка миллионов событий в сутки это страшно медленно. Это можно сделать быстрее.

Разумеется, здесь есть проблемы с безопасностью. Как только дается возможность выполнить какой-то произвольный код, надо думать о том, что будет, если добрый человек и придет и начнет рассыпать спам с нашего кластера, открывая сокеты и начиная находить какой-нибудь SNTP-сервер и так далее.

Самое главное (или не самое, но тоже существенное) — вопрос в локальности этих данных и в том, что отчеты считаются, как правило, за один и тот же период несколько раз. Есть отчеты за час, за сутки, за неделю, за месяц. Эти периоды вкладываются друг в друга. Соответственно, нам эту функцию приходится пересчитывать в таком случае 4 раза. Это плохо и нерационально.

Решение в реальном мире

- Аналогично работе с сегментами, разделим процесс на 2 этапа:



Что мы предлагаем в этом месте. Сделать примерно то же самое, что и с сегментами. Ввести дополнительную сущность в вычисленные значения, отделить в отдельный этап вычисление пользовательской функции. Расчет пользовательских отчетов проводить уже на основе логов и этих предвычисленных значений.

Этапы расчёта: первый этап

- Производится как можно чаще (например, раз в час)
- Расчёт значений вынесен в отдельный job — легко контролировать injection, устанавливать лимиты используемых ресурсов, выделять по JVM каждому
- Пользовательский Java-код по необходимости на лету компилируется с помощью Janino

Что в этом хорошего. Производить первый этап как можно чаще. Он ни от чего не зависит.

Расчет этих самых значений вынесен в отдельный job. Тем самым мы можем (насколько это возможно, насколько мы захотим это делать), полностью контролируя эту штуку, выставлять лимиты используемых ресурсов, выделять JVM соответствующие security, засовывать ее в какую-то виртуальную машину и так далее.

Пользовательский код, который приходит на языке Java, мы по необходимости на лету компилируем с помощью компилятора — в нашем случае Janino.

Этапы расчёта: второй этап

- Производится по необходимости
 - Как правило – за отчётный период – раз в сутки, в неделю, в месяц и т.д.
- Только выбирает и использует нужные заранее вычисленные значения
- Не имеет никакого code injection
- Имеет гарантированную алгоритмическую сложность и чётко прогнозируемое время выполнения

Второй этап производится по необходимости. Рассчитывается за отчетный период. Выбирает и использует только нужные заранее вычисленные значения. Совершенно чист от всяких code injection. Имеет гарантированную (это важно) алгоритмическую сложность и четко прогнозируемое время выполнения. Все уже рассчитано, никаких сюрпризов в данном месте нет.

Обеспечение безопасности выполнения чужого кода

- Каждому счётчику – отдельный mapper ⇒ выделенный процесс JVM
- Выполнение кода с максимально урезанными правами под Java security manager
- Контроль за потребляемыми ресурсами средствами ОС
- В перспективе – можно реализовать полноценную виртуализацию отдельных JVM

Как мы обеспечиваем безопасность выполнения чужого кода. На самом деле, совершенно банально. Большую часть здесь берет на себя Java. Мы выделяем на каждый счетчик за счет того, что у нас это выделено в отдельный этап расчета, отдельную JVM, там выставляем Java security manager соответствующий.

За потребляемыми ресурсами ведем контроль средствами операционной системы. В перспективе можно реализовать и запуск соответствующих JVM в полностью нормальной виртуализации.

Потенциальные угрозы

- Выполнение "опасных" действий (открытие сокетов, локальных файлов, запуск процессов, доступ к среде)
 - ⇒ SecurityException
- Перерасход ресурсов JVM
 - ⇒ JVM аварийно завершается с OutOfMemoryException
- Перерасход ресурсов процесса ОС (процессорное время)

Выполнение опасных действий тем самым ловится JVM с SecurityException'ом. Перерасход ресурсов упирается, как правило, тоже в какие-то Exception JVM. Перерасход ресурсов процесса операционной системы (в частности время выполнения) просто приводит к тому, что JVM убивается. Соответственно, мы не считаем этот отчет для данного конкретного пользователя.

Пример пользовательского кода

```
public class SectionFirstPathComponent extends RST {
    @Override
    public void process(TupleEntry in, CMap out) {
        String pagePath = in.getString(F_PAGE_PATH);
        long flag = in.getLong(F_FLAG);
        if (pagePath.length() >= 1
            && ((flag & (FLAG_PAGE_EXTERNAL | FLAG_NOT_PAGE_VIEWS)) ==
0L)) {
            int p = pagePath.indexOf('/', 1);
            if (p == -1) {
                out.put("section", pagePath.substring(1));
            } else {
                out.put("section", pagePath.substring(1, p));
            }
        }
    }
}
```

Так выглядит пример пользовательского кода. Тут, в общем, смотреть неважно и неинтересно. Единственное, что здесь надо отметить, это банальный интерфейс, в котором есть всего одна функция — процесс. У нее есть вход, где она получает строчку логов с именованными значениями на вход и выход с такими же именованными вычислennыми значениями.

Настройка пользовательского отчёта

Настройка самого отчёта, т.е.:

- что группировать
 - что агрегировать
 - как агрегировать
 - как отображать результат в веб-интерфейсе
- реализуется в виде простого YAML/JSON файла.

Настройки пользовательского отчета, соответственно, говорят нам на втором этапе, что группировать, что агрегировать, как именно агрегировать и как отображать эти результаты.

Пример описания пользовательского отчёта

```
ecommerce.geo.region:
    gb:
        - geo.country
        - geo.region
    sum:
        - order.total
    cd:
        - order.id
```

Заказывается GroupBy по 2 полям

Заказывается сумма поля order.total

Заказывается COUNT DISTINCT поля order.id

Настройку пользовательского отчета мы делаем в виде JAMA. Очень любим, с одной стороны. С другой стороны, глубоко ненавидим JAMA. Все сводится к тому, что есть идентификатор отчета. Мы задаем группировку, некий GroupBy по полям. Задаем агрегацию, в данном случае сумма. Также какой-то COUNT DISTINCT по каким-то полям.



Таким образом, целиком схема вместе с сегментами и пользовательскими функциями выглядит следующим образом. Это довольно тривиальная конкатенация двух предыдущих схем.

Что в итоге получили пользователи?

- Сегменты могут создавать все пользователи из удобного конструктора в веб-интерфейсе
- Продвинутые пользователи могут сами настраивать себе отчёты практически произвольной сложности, написав несколько строчек кода на Java и описав вид желаемого отчёта
- Менее продвинутым пользователям это поможет сделать службу внедрения

В итоге пользователи получили возможность создавать сегменты в веб-интерфейсе. Продвинутые пользователи могут писать строчки кода на Java и получать те отчёты, которые надо. Менее продвинутым пользователям это может помочь сделать службу внедрения.

А что получили мы?

Общий объём пользовательских отчётов ежедневно:

- 100 мегабайт
- 1.2 млн строчек
- ~10000 отчётов

Мы получили мегабайт отчетов (это уже агрегированные данные) в день. 1,2 миллиона строчек, примерно 10 тысяч отчетов у нас сейчас этой технологией пользуются.

Наша экономия

- Обычный отчёт за сутки считается за время от ~3 до ~30 машиночасов^{*} в зависимости от сложности
- Все пользовательские отчёты считаются за один проход длиной 32-35 машиночасов^{*} в сутки
- $10000 \times (3..30) = (30000..300000) \text{ vs } (32..35)$
- Итого – экономия на 3-4 порядка

Экономия, которую мы получили. По совершенно простым подсчетам — это 3-4 порядка. Если бы мы запускали все эти отчеты традиционным способом, это занимало бы порядка 300 тысяч машиночасов. Мы это все считаем сейчас за порядка 30-35 машиночасов в сутки.



Все. Спасибо большое! Есть какие-то вопросы?

Вопросы и Ответы

Вопрос из зала: Спасибо за доклад. Скажите, вы каким-то образом храните исходные данные? Допустим, логи. Если да, то за какой период. Построив отчет, пользователь сможет видеть историю на год назад, прошлый год и так далее?

Михаил Якшин: Это достаточно подробно в докладе на предыдущей конференции рассказывалось. Мы совсем не храним исходные логи. Мы храним чуть-чуть процессинговые логи. Логи мы храним в течение, по-моему, минимум трех месяцев (по соглашению — больше). Отчеты мы сейчас храним безвременно. В основном, всех интересуют отчеты, а не логи.

Вопрос из зала: Отчет — это в смысле агрегация?

Михаил Якшин: Да. Спасибо.

Архитектуры Backup & Recovery решений

Илья Космодемьянский



Илья Космодемьянский: Добрый день, уважаемые коллеги. Меня зовут Илья Космодемьянский. Я буду рассказывать про то, как заниматься Backup и Recovery в самых разных системах. Вообще я больше работаю сейчас с Telecom'ом, но с web'ом, в принципе, тоже. Я думаю, что довольно широкий набор каких-то таких базовых вещей и каких-то конкретных рецептов я тут сейчас расскажу.

Откуда вообще возникла идея про это рассказывать, и зачем. Казалось бы, что такое Backup, знают все. (Обращаясь к слушателям). Кто не знает что такое Backup? Есть хотя бы один такой человек? Да. Не верю.

С другой стороны, формально к этому вопросу не подходят, потому что вроде все знают. Люди вообще формулировать что-либо особо не любят. Зачем очевидные вещи 33 раза пересказывать. В то же время там методический подход местами очень важен, потому что штука достаточно критичная.

Если мы будем говорить о Backup'е, то надо сразу сказать, от чего мы защищаемся и чего хотим в результате получить.

Падения неизбежны

- Не бывает абсолютной отказоустойчивости
- “Быстро поднятое не считается упавшим”

Случается так, что софт у нас падает. Абсолютно надежного софта, во-первых, просто не бывает. Если кто-то вам говорит, что вы можете собрать какую-то мегаферму из кучи серверов, такую, что она никогда не упадет, вам, скорее всего, врут. Вкладывать огромные деньги в такую инфраструктуру, которая будет десятикратно продублирована, достаточно глупо.

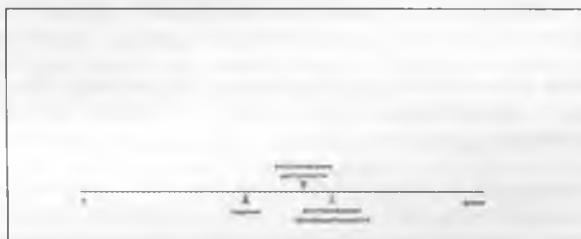
В результате, да, она будет десятикратно продублирована. Потом выяснится, что неправильно закоммутировали Cisco, за которой она стоит. От этой всей надежности не останется более или менее ничего. Стандартный подход в таких решениях — это давайте делать софт таким образом, чтобы его можно было поднять как можно быстрее и восстановить его работоспособность.

“Поднятая” система

- Доступна
- Восстановлена производительность
- Не потеряны данные

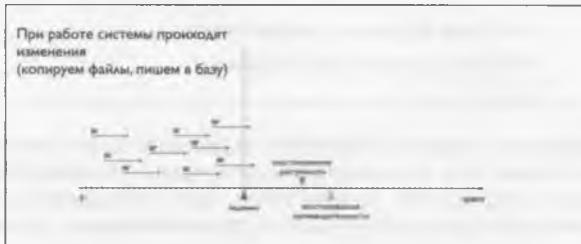
Что такое «поднять» после аварии, после падения какого-то.

- Это сделать систему снова доступной, чтобы она отвечала на запросы пользователей, чтобы она как-то осуществляла то, что она делала до падения.
- Восстановить прежнюю производительность этой системы, как до падения было. Может быть, это иногда не сразу происходит.
- Самое, наверное, важное — что нужно не потерять данные, потому что обычно с данными что-то происходит. Весь наш web — на один порт пришел запрос, с этого порта пришел ответ. Это какой-то обмен данными. В любом случае, они откуда-то берутся.



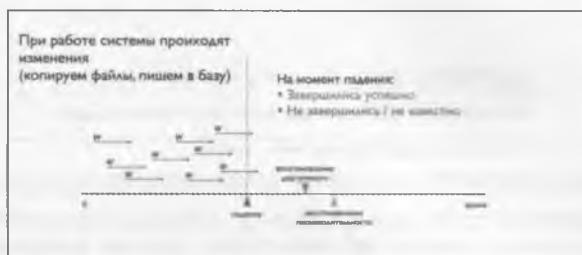
Если я нарисую временную линию, где у нас работает система, и в какой-то момент у нас происходит падение, то пройдет некоторое время, прежде чем мы сможем восстановить доступность. Возможно, немного позже мы будем восстанавливать производительность. Может быть, у нас это все произойдет одновременно.

Если это, например, статический сборник каких-то графических файлов, то вопрос вообще достаточно небольшой. Допустим, у нас просто вытащили электрический кабель, у нас все упало. Но мы включили его обратно — все завелось, опять снова работает, и работает хорошо. Это совершенно нормальная ситуация если у нас нет никакого изменения.



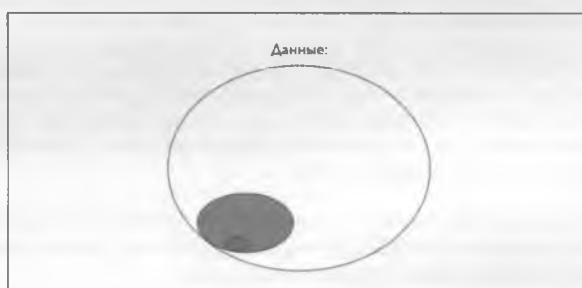
На практике у нас, естественно, изменения есть. Мы можем копировать файлы, пользователи могут нам заливать какие-то файлы, мы пишем что-нибудь в базу. Занимаемся, в общем, какими-то операциями изменения данных. Как правило, это все равно в конечном итоге работа с данными.

У меня на слайде по времени отмечены происходящие операции записи. Одни у меня, видите, зелененькие, а другие — красненькие.



Тут критерий очень простой. Если пользователь загружает файл, он полностью файл загрузил, в базе мы все это записали, все Ok, готово полностью. Это, значит, зелененькое изменение, то есть на момент падения оно полностью произошло.

Если же загрузка файла или запись какой-то длинной прстыни в базу (она происходит не в одну секунду, она происходит некоторое время). Если как раз в этот момент, когда это изменение было не завершено, происходит падение, то у нас данные какие-то, возможно, битые. Пользователь загружал нам файл, а оказалось, что загрузил полфайла, и мы вообще не знаем что у него во второй половине. Либо дневной баланс подводили в какой-нибудь платежной системе, а оно посередине взяло и упало.



То, что я сейчас объяснил, можно посмотреть на этой диаграмме. Большой белый круг — это фактически все наши данные, которые не менялись, можно спокойно систему поднять, и ничего не произойдет. Есть зеленые данные, про которые тоже можно сказать, что ничего не произойдет, если мы систему поднимем. Все изменения произошли, и фактически это все равно, что мы эти данные не трогали в большинстве случаев.

Маленький красненький кружочек — это ровно те данные, с которыми активно происходила какая-то запись. Когда мы поднимаем систему, нам нужно хотя бы понять, что вообще произошло. Целые они, не целые, что с ними делать.

Процедура восстановления

- Восстановлены данные, которые не изменились
- Восстановлены данные, которые изменились в момент падения
- Восстановлена работоспособность системы

Логично, что для подъема системы нужна какая-то процедура восстановления. Фактически мы восстанавливаем данные, которые не изменились, данные, которые изменились. После этого мы можем как-то нашу систему запускать и дальше с ней особым образом развлекаться в ожидании другого падения.

Восстановлены данные, которые изменились в момент падения?

- Обычно это означает, что изменения вычищены из системы
- В транзакционных системах: откатить изменения

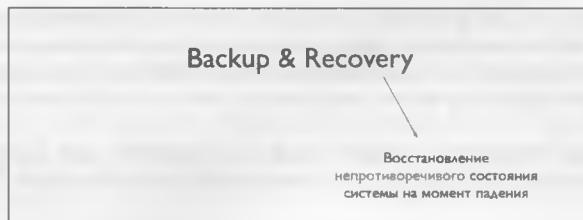
Если мы говорим о восстановленных данных, то, как правило, это следующее. Допустим, та же самая ситуация: мы заливали какие-то большие файлы (графические или видео), например, на хостинг. Если залилась половина файла, мы, скорее всего, не знаем что дальше продолжать делать и, скорее всего, эти вещи вычищаем из системы. Иначе у пользователя будет лежать битый файл, наполовину квадратиками.

В принципе, если мы будем очень умными, может быть, напишем какую-то систему загрузки чанками и будем где-то учитывать, сколько чанков загрузилось, будем предлагать ему это продолжить. Но это уже из области очень специальных задач.

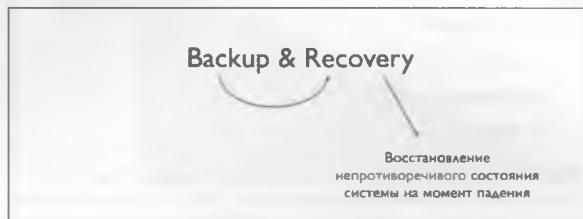
В транзакционных системах обычно про такую очистку говорят, что «изменения откатываются». На самом деле, с транзакционными системами все немного сложнее, потому что одни изменения могут накатываться, другие откатываться в зависимости от того, что там побилось при падении. Об этом мы сейчас еще поговорим.

Восстановление
непротиворечивого состояния
системы на момент падения

Итак, мы видим главную цель процедуры восстановления как таковой. Нам надо систему восстановить в логически непротиворечивом виде. Чтобы все файлы, которые загружены, были загружены. Все транзакции, которые происходили, должны быть либо закоммичены, либо откачаны. Непосредственно перед моментом падения. Это основная цель, для чего мы все делаем.

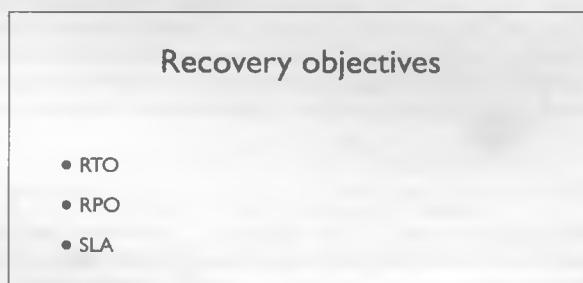


Грубо говоря, я провожу на слайде стрелочку от Recovery к этой вот цели. Мы делаем recovery, чтобы добиться этого.



Если мы будем говорить о Backup'е, то главная идея Backup'а – это обеспечить Recovery всем необходимым. Очень часто я наблюдаю такую картину, что вроде положено делать Backup – мы его делаем. Как мы будем с него восстанавливаться – это уже... Упадем – будем думать.

На самом деле, это в корне неправильный подход. Backup предназначен для того, чтобы восстанавливаться. Соответственно, нужно заведомо об этом думать и, исходя из этого, его планировать. Главное слово идет первым, потому что по алфавиту первое или благозвучнее так, наверное. Но Recovery, на самом деле, гораздо важнее.



Говорят о так называемых целях восстановления (по-английски Recovery objectives) и обычно выделяют 3 основные категории.

- Recovery Time Objective, когда мы стремимся сократить минимально время недоступности системы. Система у нас как-то жестко связана по производительности, ее простои – это огромные потери для бизнеса. Мы, соответственно, хотим, чтобы она как можно быстрее восстановилась.
 - Есть такой до некоторой степени антагонистический подход, а именно RPO (или Recovery Point Objective), когда мы стараемся добиться максимального восстановления данных, то есть потерять как можно меньше данных.
- Эти два подхода конфликтуют, но обычно приходят к компромиссу и стараются добиться разумного времени с максимальным сохранением данных.

- Третий вариант — так называемый SLA (Service Level Agreement). Вендор продает кому-нибудь какую-нибудь систему и говорит: «Если она упадет, мы вам ее поднимем за час». Обычно в ситуации с web-проектом такое не всегда проходит, потому что web — это все-таки вещь очень динамичная, его вендоры «под ключ» не делают.

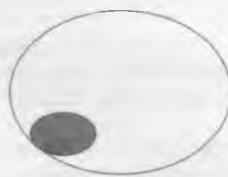
В данном случае можно неким SLA считать наши соображения, как выбрать между Recovery Time Objective и Recovery Point Objective, но, в основном, нас интересует именно комбинация первых двух.

Что бэкапить?

- Всё?
- Системный раздел ОС?
- Данные?
- Важные конфиги?

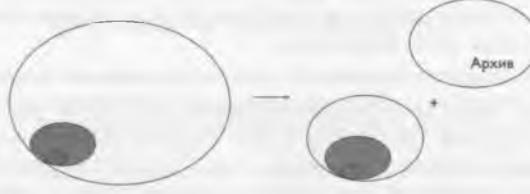
Дальше важный момент — понять, что мы собираемся бэкапить. Естественно, лучше всего бэкапить все, но это занимает очень много места. Как правило, бэкапам подвергаются данные и какие-то системные разделы операционной системы. В принципе, эти две вещи в результате дают систему в работоспособном состоянии. Здесь важно не упускать мелочей, о чём я еще немного позже отдельно скажу.

Что бэкапить?



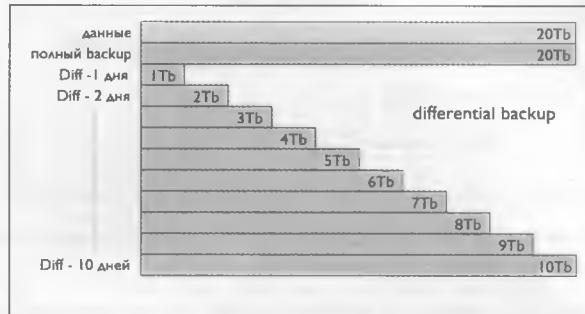
Если мы представляем наши данные с той картинки. Допустим, нас действительно такая хорошая картина, а именно у нас очень небольшое количество данных постоянно изменяется, много данных таких лежит.

Что бэкапить?



В этом случае, в принципе (чтобы понимать), часть данных у нас реально архивная. Если мы не можем вытащить ее в какой-то архив, чтобы не гонять большое количество данных, в любом случае, мы можем как-то разобраться что бэкапить, что не бэкапить.

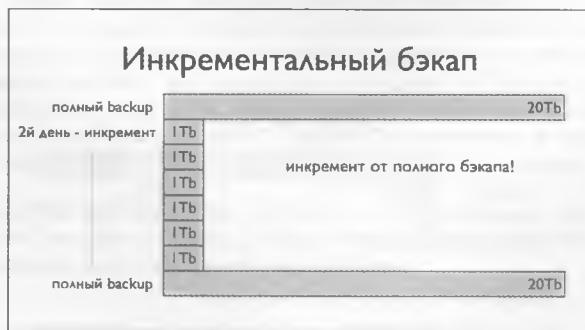
По идее, наша задача — хорошенько сократить эту массу данных, потому что Recovery Time Objective напрямую от этого зависит. Если у нас бэкап 20 терабайт, и мы всегда восстанавливаем 20 терабайт, все могут посчитать, сколько займет, например, перекачивание такой штуки по сетевому диску.



Давайте поговорим непосредственно о подходах к бэкапу.

На слайде зеленым обозначен наш DataSet, который мы хотим бэкапить. У нас есть данные, их 20 терабайт. Допустим, в первый день (обычно это стараются делать с субботы на воскресенье) мы делаем полный бэкап. Каким-то образом копируем все 20 терабайт. Этот подход называется «дифференциальный бэкап». Каждый день мы будем добавлять некий difference между full бэкапом и тем, что у нас записалось в базу.

Как мы видим по этой диаграммке, где возрастает количество места, это не всегда оптимально, потому что уже через 10 дней этот дифференциальный бэкап будет составлять половину всего DataSet'a. За счет того, что на слайде показано белым цветом, мы, конечно, получаем экономию места. Но если посчитать все эти серенькие пустые клеточки, то места занимается реально очень много. Фактически мы очень многие данные пишем туда по несколько раз. На самом деле это, наверное, не очень хорошо, поэтому народ обычно придумывает какие-то другие способы.

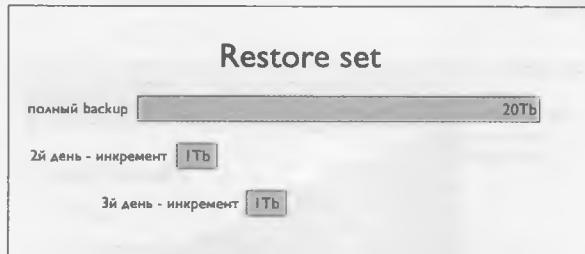


Следующий вариант — это так называемый «инкрементальный бэкап», который совершенно очевиден тоже по подходу.

Мы делаем полный бэкап, допустим, опять же с субботы на воскресенье, а потом делаем в последующие дни инкременты. Инкременты у нас происходят от полного бэкапа, и пишутся

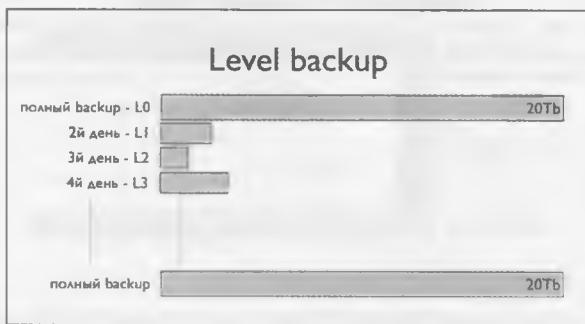
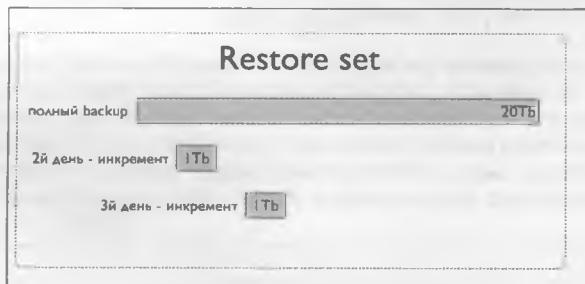
только реально те изменения, которые произошли в этот день. Потом через какое-то время (например, через неделю) мы повторяем полный бэкап.

Какие здесь есть некие подводные камни. Мы помним, что мы делаем бэкап для того, чтобы произвести restore. Чтобы произвести restore, нам нужно некоторое количество этих файлов. Допустим, мы упали на третий день



Этот restore set, который мы хотим использовать для процедуры восстановления, иначе она просто не пройдет, будет составлять 3 файла.

Если мы упали, допустим, не в среду, а в субботу утром, то нам нужно будет очень большое количество файлов для этого использовать. Результат следующий: теоретически любой бэкап может быть побитым. Успех нашего restore целиком и полностью зависит от того, не окажется ли один файл чуть-чуть битым. В результате нам нужно держать действительно очень широкую штуку для того, чтобы осуществить любое восстановление, и каждый раз молиться о том, чтобы оно не пришло битым.



Обычно используют такую модификацию инкрементальных бэкапов, называют ее level backup. Например, полный бэкап у нас отвечает нулевому уровню, и уровни дальше идут до 9-го. Каждый последующий уровень — это инкремент от предыдущего уровня.

Таким образом, мы не бэкапим файлы дважды, как, например, в каком-то простом инкрементальном бэкапе или дифференциальном. Зато получаем все изменения в гораздо более легковесном виде. Через некоторое время обычно у нас происходит полный бэкап уже в самом конце.

На практике редко используют все эти уровни бэкапа. На практике обычно используют, например, нулевой и первый. Делают большой бэкап, потом делают инкрементик, потом, соответственно, каким-то образом пишут Diff между ними уже третьим способом (например, архивные логи в базе данных).

Backup retention							
	план.	инк.	инк.	инк.	инк.	инк.	инк.
1 нед	20Tb	1Tb	1Tb	1Tb	1Tb	1Tb	1Tb
2 нед	20Tb	1Tb	1Tb	1Tb	1Tb	1Tb	1Tb
3 нед	20Tb	1Tb	1Tb	1Tb	1Tb	1Tb	1Tb
4 нед	20Tb	1Tb	1Tb	1Tb	1Tb	1Tb	1Tb
5 нед	20Tb	1Tb	1Tb	1Tb	1Tb	1Tb	1Tb
6 нед	20Tb						
	120Tb	5Tb	5Tb	5Tb	5Tb	5Tb	150Tb

Важный момент — сколько и какой бэкап хранить. Называют это словом *Retention policy*. Обычно рассчитывается как. Например, у нас в компании есть требование — мы хотим хранить бэкап 4 недели, то есть у нас *Retention* период должен быть 4 недели.

Я написал схемку — когда у нас происходит полный бэкап, когда инкрементальный. Будем для красоты считать, что инкрементальные бэкапы у нас занимают по одному терабайту, а полный занимает 20 терабайт.

Здесь написано почему-то не 4, а 6 недель. Это, на самом деле, очень важный момент. Фактически каждого бэкапа (каждого инкрементального и каждого полного) мы должны держать 4 копии, если у нас политика *Retention*'а 4 недели, потому что каждый из них происходит раз в неделю. Во вторую неделю в понедельник происходит такой-то бэкап (полный, допустим), во вторник — инкрементальный.

Если мы, например, будем держать ровно 4 недели, то получится, что у нас в любой момент времени не будет четырех бэкапов каждого вида. Если мы будем держать 5 недель, то они у нас, наконец, появляются. Но мы еще не можем удалять полный бэкап, который мы произвели в первую неделю, потому что нам нужен еще один, чтобы выполнялось условие, что в каждый момент времени у нас есть все 4. Поэтому нам нужно фактически в четвертую неделю произвести первый бэкап нулевого уровня для системы, и тогда только мы сможем удалять первую строчку бэкап первой недели.

Здесь приведена простая арифметика, сколько это занимает по месту. Бэкап — это, на самом деле, довольно большие объемы, если мы его делаем именно по-правильному и по-правильному разбираемся, в какой момент его удалять.

Backup retention							
	ПЛН.	ИНК.	ИНК.	ИНК.	ИНК.	ИНК.	ИНК.
1 нед	20Tb	1Tb	1Tb	1Tb	1Tb	1Tb	1Tb
2 нед	20Tb	1Tb	1Tb	1Tb	1Tb	1Tb	1Tb
3 нед	20Tb	1Tb	1Tb	1Tb	1Tb	1Tb	1Tb
4 нед	20Tb	1Tb	1Tb	1Tb	1Tb	1Tb	1Tb
5 нед	20Tb	1Tb	1Tb	1Tb	1Tb	1Tb	1Tb
6 нед	20Tb						
	120Tb	5Tb	5Tb	5Tb	5Tb	5Tb	150Tb

Напоминаю, что это у нас будет период 4 недели.

Бэкап файловой системы

- dump/restore
- cpio и tar (GNU!!)
- dd
- rsync

Если мы делаем просто бэкап файловой системы, в каких случаях это надо. На самом деле, данные пишут кто куда хочет. Обычно это пишут в базу данных, кто-то любит писать в Excel (в смысле в NoSQL). Бэкапить надо все-таки какой-то системный софт или, например, бинарное хранилище.

Для этого используется ряд утилит, которые я написал на слайде.

- Первая из них — это dump/restore, довольно древняя штука. Я думаю, большинство используют Linux все-таки в продакшне (как-то в последнее время так повелось), вряд ли какой-нибудь Solaris или на HP UX.
- У Linux'a с dump'ом есть некоторые проблемы. Он, в общем, не особо рекомендован к использованию. Плюс к тому он обладает рядом тонкостей по части кривых рук.
- Поэтому, наверное, проще использовать cpio или tar. В обоих случаях надо использовать GNU'шные версии, а не нативные утилиты, потому что они гораздо удобнее и лучше работают. Что конкретно — это уже дело вкуса. Но бэкапить этим 20 терабайт я бы, честно говоря, не рискнул. Это надо будет делать отдельный кластер, выполняющий бэкапы, а потом делать бэкап этого кластера.
- Часто принято делать системный раздел. Он, например, небольшой, и можно его чисто dd скидывать раз в сутки на новый диск. Если сервер у нас вылетел, то мы, соответственно, заменяем этот диск и взлетаем. Так очень удобно делать для серверов, например, не баз данных, а если у нас есть какие-нибудь backend'ы, frontend'ы, у которых, в общем, довольно типовая инсталляция. Залили, подключили, данные на них пошли.
- Стандартный подход — это использование rsync'a. Rsync — это как таковая не бэкапная утилита. Мы можем сделать бэкап с помощью cpio или с помощью tar'a, а rsync'ом мы просто его передать на какой-то другой сервер. Естественно, если мы имидж храним в том же месте, то от падения сервера нас ничто не защищает.

Базы данных

- Заточены под backup&recovery
- Холодный и горячий backup

Гораздо интереснее обстоит дело с базами данных, как, наверно, догадались. Нормальная база данных — это система, в которой все предназначено для того, чтобы данные были сохраны. Такая старая DBA'шная мудрость, что DBA может, грубо говоря, SQL знать со словарем. Он может в справочнике смотреть каждый раз какую-то команду. Но весь процесс Backup и Recovery он должен знать просто на зубок, потому что иначе он профнепригоден.

В базах данных различают горячий и холодный бэкап. Холодный — это когда мы базу выключаем, файлы копируем. В большинстве случаев это хоть как-то работает. Обычно применяется когда, во-первых, мы можем себе позволить какой-то downtime, а во-вторых, когда, например, нужно смигрировать на новую версию.

Обычно же база работает в достаточно напряженном режиме. Поэтому хочется обычно горячего бэкапа, когда мы не гасим базу и желательно еще при этом по минимуму как-нибудь проигрываем с нагрузкой. Большинство баз предоставляют нормальный горячий бэкап.

Архивирование WAL



Что я имел в виду, когда говорил, что база данных предназначена для резервного копирования и последующего восстановления. В любой разумной базе есть Write-Ahead Log — это некое последовательное описание изменений, происходящих в базе. В упрощенном виде в нем, в каждом его сегменте имеется undo и redo записи. По redo записи мы можем накатить заново транзакцию, которая у нас отвалилась в процессе падения, по undo записи откатить ее назад, если она уж совсем оставляет данные в неконсистентном виде.

Простая идея — давайте мы будем брать этот лог, поскольку в нем пишутся все изменения, которые нас интересуют, и архивировать. Обычно этим занимается отдельный процесс, он архивирует лог и куда-нибудь складывает. После этого мы можем эти логи куда-нибудь копировать. По ним потом в обратном порядке будет проходить процедура restore, что-то накатывать, что-то откатывать.

Как выглядит бэкап?

- Копируем dataфайл
- Он (возможно) неконсистентен
- Для восстановления нужны архивные логи

Как будет выглядеть в такой ситуации алгоритм. В любой базе есть какой-то dataфайл. Мы берем и копируем его. Обычно базу данных надо включить в специальный режим для этого. В каких-то случаях это делается специальными командами, в каких-то случаях — специальным процессом. Файл копируется с отслеживанием, трекингом этих самых изменений, которые там происходят.

В результате у нас получается неконсистентный файл (то есть логически нарушенный). Мы можем его докатить с помощью этих логов, которые все это время писались, до консистентного состояния. Для `restore set`, который нам нужен для восстановления, это фактически неконсистентный файл с данными и архивные логи для того, чтобы его восстановить в правильное состояние.

Oracle

- RMAN
- Уровни
- параллелизм
- работа с железом
- каталог

Как это устроено у Oracle, для того чтобы как бы понимать о чем идет речь. У Oracle этим занимается отдельный процесс. Он поддерживает различные уровни бэкапа — те уровни, о которых я говорил. Если у нас есть 20-терабайтная база, мы можем существенно экономить на хранении.

Он поддерживает распараллеливание как процедуры бэкапа, так и процедуры восстановления. Почему это очень важно — потому что когда мы бэкапим, мы не хотим слишком долго насиживать базу этой операцией. Нам интересно сделать ее быстрее.

Когда у нас база 20 терабайт, то она будет просто очень долго бэкапиться, на самом деле, и параллелизм там нужен уже просто для того, чтобы успевать делать бэкапы с должной частотой. Упаковать мы можем в любой момент, а у нас, например, бэкап делается два дня. Будет очень плохо, если мы в эти же два дня и упадем, потому что мы потеряем очень большое количество данных. У нас будет совершенно не выполнена Recovery Point Objective.

Он умеет работать с железом, он сам умеет работать с ленточными библиотеками и, что не маловажно, он обладает каталогом. Каталог — это запись того, где есть какие бэкапы, по которым процедура Recovery может ориентироваться, что дальше должно происходить.

PostgreSQL

- Полный бэкап
- Архивирование WAL
- Нет уровней (вернее есть но один)

В Postgres'е, естественно, с этим все попроще, но идея более или менее та же. Postgres умеет делать полный бэкап. Делает он его в неконсистентном виде. Вы сначала запускаете процедуру. Когда бэкап стартовал у вас, копирование средствами файловой системы произошло, вы говорите, что он остановился. У вас архивные логи, которые будут за это время нужны, помечены, и архивная команда, которая в `recovery.conf` прописана, берет эти логи, копирует, и вы можете все это дело восстановить.

Из ощутимых недостатков — не хватает явно уровней. Представляете, вы в субботу вечером делаете `level 0 backup`, а потом нагоняете это дело логами. Если вы падаете в пятницу вечером, например, то вам может потребоваться очень и очень много времени для того, чтобы все эти логи накатить. Просто это будет очень длительная штука.

Если бы у вас были какие-то бэкапы другого уровня, которые бы вы, например, каждую ночь каждого буднего дня проводили, то это занимало бы существенно меньше места. Самое главное — восстановление происходило бы гораздо быстрей. Нет параллелизма — это чисто проблема по скорости уже, получается.

MySQL

- Enterprise Backup и Percona XtraBackup
- Только для транзакционных энджинов

Если мы будем говорить о MySQL, то в MySQL недавно появился бэкап, а именно это Enterprise Backup и Percona XtraBackup. Рассматривать первый я, честно говоря, считаю странным, потому что лицензия стоит порядка пяти килобаксов за один сервер. Если тратить такие деньги на такую лицензию, то уж можно купить DB2 Workgroup Edition и поиметь гораздо больше плюшек, чем MySQL, да еще и за деньги.

Естественно, работает он только для транзакционных энджинов. Почему я, собственно говоря, рассказывал про redo log. Практически это лог наката транзакций. Соответственно, если транзакций нет, то консистентный бэкап для нас получить очень сложно.

В принципе, Percona'вский бэкап поддерживает MyISAM, но с одной маленькой оговоркой — он требует глобальной блокировки на все это дело. InnoDB бэкапит нормально, а MyISAM блокирует. Соответственно, если у вас большое количество данных в MyISAM'и, и вы выставляете глобальную блокировку на него, понятное дело, что, наверное, это не всегда совместимо с операционными необходимостями.

Вредные советы

Заменяйте бэкап репликацией!

- Падения бывают не только от сбоя железа, но и от кривых рук
- Проблемы от кривых рук очень быстро расползаются с мастера
- За самой репликацией нужен глаз да глаз

Напоследок я расскажу некое количество вредных советов — чисто из практики, то, с чем сталкивался.

Первая идея. У нас 26 реплик MySQL, поэтому бэкап нам не нужен, какая-нибудь реплика да выживет. Основная проблема здесь и самая очевидная — это то, что, в принципе, все падения бывают либо аппаратные, либо логические. И те, и другие могут возникать не только от каких-то объективных причин (попали на какой-то баг ядра Linux'а или отлетел жесткий диск), они бывают от кривых рук.

То, что Петя передо мной сейчас рассказывал, что мы можем делать отложенную репликацию в MySQL'e, потому что кто-то дропнул таблицу, и оно с master'a поехало везде. Ровно на этот случай как раз репликация — это очень плохая идея. Лучше иметь все-таки еще и backup. Репликация может отъехать, за ней надо мониторить, внимательно смотреть. Если у вас пошел drop table на master'e, то надо еще успеть поймать, добежать, разобраться, что с этим сделать.

В принципе, оно должно быть все вместе, то есть должен быть и бэкап, и репликация. Я уж не говорю о том, что в стойку может попасть метеорит, и вообще хорошо было бы иметь бэкап где-нибудь еще в другом месте.

Вредные советы

Выделяйте человеческие ресурсы на обеспечение бэкапа по остаточному принципу!

- Пока не понадобилось восстановление, про бэкап никто не помнит
- Часто это обязанность самого младшего админа

Следующий вредный совет. На производство бэкапа ресурсы надо выделять по остаточному принципу. Обычно этим занимается самый младший сисадмин, или эту дополнительную почетную обязанность на сильно занятого сисадмина. Это очень плохая практика. Бэкапом нужно заниматься, подходя к нему со знанием дела и понимая, что весь бизнес компании может стоять под угрозой, если вдруг что-то не восстановится.

Не надо объяснять, что даже если мы не являемся банком с транзакциями, у нас есть пользователи, которые уйдут к конкурентам немедленно после того, как у нас все вылетит на некоторое время, и мы будем очень долго восстанавливать работоспособность. Вспомните историю с DDoS'ом ЖЖ, и какие графики люди публиковали про то, сколько народу уходит на «Facebook». По всем оценкам счетчиков огромное количество людей после таких outage'ей размером в день просто валило.

Вредные советы

Тестовое восстановление придумали трусы!

- Любой бэкап может быть битым
- Проверить можно только восстановлением

Еще один вредный совет. Тестовое восстановление придумали трусы. Любой бэкап, какую бы процедуру валидации мы на него ни повесили, может быть битым. Проверить, что бэкап не битый можно одним способом — восстановиться. В конечном счете именно процедура восстановления показывает, что происходит.

В регламенте любой системы должно быть предусмотрено тестовое восстановление, например, раз в неделю. Раз в неделю мы проверяем те бэкапы, которые у нас есть, на какой-нибудь более хиленькой машинке не с хорошим storage'ем, а с каким-нибудь подключенным NAS'ом. Мы ее восстанавливаем и только после этого можем у себя в каталоге пометить, что этот бэкап у нас валидный.

Вредные советы

Главное чтобы в целом все как-то работало!

- С backup & recovery мелочай не бывает
- "В системе было предусмотрено решение HP DataProtector стоимостью \$\$\$\$\$\$, но никто не вспомнил настройки LVM"
- Сервер с каталогом RMAN восстановить не удалось

Еще один вредный совет. Мы как-то все сделаем, вроде у нас все работает, но мало ли, каких-то мелочей у нас не будет. Это неправда. Я говорю: Backup и Recovery — это процедура очень важная, и мелочей там нет.

(Вопрос к залу). Кто-нибудь использует Linux'овый LVM? Народ использует Linux'овый LVM. Знаете про команду dg_cfg_backup? Уже меньше. Как часто вы ее используете? Кто-нибудь использует ее хотя бы раз в несколько дней или хотя бы на каждый setup, когда изменяется группа? Вы видите, насколько количество народа сокращается.

В реале я, например, наблюдал систему, когда все забыли, как была устроена конфигурация LVM'a, не сохранили эти диски, и, не смотря на наличие хорошего... Ладно, хорошего, нехорошего, но дорого коммерческого бэкапного софта, восстановление не прошло просто потому, что забыли разделы на одном системном диске. После этого данные восстанавливали таким способом очень долго.

Следующая проблема. У нас есть какой-то каталог, где мы храним, где у нас лежит какой-то бэкап (например, Oracle'овый, RMAN'овский каталог). Если он лежит на том же сервере, который мы собираемся восстанавливать, это плохо, потому что он у вас упал, и что с этим делать, совершенно непонятно. У вас есть набор бэкапов, а как его восстанавливать — неясно.

Надо хорошенъко заботиться о том самом бэкапе бэкапов, когда мы выносим на какой-нибудь отдельный сервер или на бумажке выписываем себе, какие бэкапы у нас есть, в каком порядке процедура восстановления у нас должна работать.

Вообще порядок процедуры восстановления надо знать просто идеально. Когда случается проблема, во-первых, велик риск сделать что-нибудь не так, просто ошибившись, а во-вторых, просто не понять, что сделать. От этого начинаются серьезные грабли.

- [W. Curtis Preston - Backup & Recovery Inexpensive Backup Solutions for Open Systems](#)
- [Steven Nelson - Pro Data Backup and Recovery](#)
- [Oracle Backup and Recovery User's Guide](#)

В завершении, если кто пожелает ознакомиться подробно с разными бэкапными архитектурами и решениями, которые вообще в мире бывают, есть 3 такие книжки. Последняя — это документация по Oracle. Две книжки про бэкапную архитектуру, которые я мог бы порекомендовать. Первая книжка — вообще страшно рекомендую. Там очень много страшных историй от знакомых автора. «Я работал младшим сисадмином, меня выгнали за то, что у нас пропало то-то...». Увлекательнейше читается. Очень рекомендую.

В общем, все. Спасибо. Вопросы.

Вопросы и Ответы

Вопрос из зала: Здравствуйте. У меня такой вопрос. Допустим, у меня есть 10 инстансов Postgres'a, и они обмениваются между собой данными. Как мне сделать бэкап, чтобы можно было эту систему восстановить консистентно?

Илья Космодемьянский: Во-первых, тут большой вопрос — как они у вас обмениваются данными. Если они у вас обмениваются данными активно в обе стороны, то вообще сначала я бы вам посоветовал, потом уже думал что с этим делать. Вы имеете серьезную проблему с очень непредсказуемой репликацией, которая, на самом деле, требует большого слежения.

Вопрос из зала: Я имею в виду не репликацию, а транзакцию — перемещение данных с одного сервера на другой.

Илья Космодемьянский: То есть у вас на всех серверах разная схема, просто они обмениваются данными?

Вопрос из зала: Схема одна и та же. Чтобы понятнее: есть игра онлайн...

Илья Космодемьянский: Да, я подозревал.

Вопрос из зала: Есть персонаж. Он передает предмет другому персонажу. Этот предмет переезжает из одной базы в другую, и мне нужно сделать бэкап этой системы.

Илья Космодемьянский: В реальности вам надо делать бэкап каждой базы отдельно.

Вопрос из зала: Я понимаю, но как мне сделать консистентно? Мне нужно сделать бэкап на тот момент времени, когда предмет есть только в одной из баз.

Илья Космодемьянский: В реале у вас каждый бэкап в отдельности будет консистентный, потому что он будет сделан на некоторое время. Если вы постоянно будете делать streaming логов и включать base backup postgres'овый, то в каждой базе он будет консистентный для

нее конкретно. Другой вопрос состоит в том, что абсолютно одновременно у вас эти бэкапы происходить не будут. Будет некая проблема, что у вас юзеры в данный момент переезжали, вам нужно будет с этим делом разбираться.

Единственный практический совет, который можно дать навскидку, это делать эти бэкапы в разном расписании. Если эта база, особенно на каждой node, не совсем большая (например, до 200 гигабайт), то, в принципе, это может быть разумно. Вы как раз несколькими своими инстансами более или менее равномерно покроете сутки, например. Просто тогда вы, восстанавливая одну базу, будете точно понимать, что она у вас консистентна, а вторая, может, в другое время упадет.

Вопрос из зала: Спасибо.

Вопрос из зала: Вы говорили про бэкап данных, но какой смысл бэкапить данные, которые неуникальны? Бэкап системныхパーティций, инсталляции операционки — это все дублируется и бэкапится в репозитариях внешних.

Илья Космодемьянский: Если дублируется, бэкапится в репозитариях внешних, то значит они бэкапятся.

Вопрос из зала: Не имеет ли смысла это перегенерить? Зачем бэкапить данные, которые быстрее перегенерить, чем...

Илья Космодемьянский: Если быстрее перегенерить, то это гораздо проще. Зачем время тратить на это? Не вопрос.

Вопрос из зала: Немножко дополню, получается, предыдущего докладчика. Вы затронули вопрос восстановления бэкапа данных. Но, если вспомним, то любая приличная база данных половину данных держит в кэше и пишет на диск в очень редких случаях.

Илья Космодемьянский: Ну, да, если размер базы данных 20 терабайт.

Вопрос из зала: Это да, но все равно все свежие данные у нее висят в кэше. Насколько я знаю, только NetApp предлагает решение — они предлагают промежуточный драйвер, который при бэкапе аккуратно все скачивает с кэша.

Илья Космодемьянский: Вы знаете, как работает Write-Ahead Log?

Вопрос из зала: Приблизительно — да.

Илья Космодемьянский: Почему тогда вопрос про кэш? Может быть, мы, конечно, говорим про MySQL, но MySQL не относится к приличным базам. Любая другая база, естественно, пишет sequential log на диск (если только, например, в Postgres человек не поставил какой-нибудь `fsync_delay`). Но залог ее заточенности под то, что процедура Recovery успешно будет происходить, заключается именно в том, что прежде чем в блочное хранилище, в dataфайл попали все изменения, прежде всего, они записаны sequential log.

Если, например, вы будете наблюдать Oracle, у которого заканчивается место в этом самом логе, который пишется, или он занят архивированием в данный момент, то Oracle у вас перестанет принимать пишущие транзакции и будет об этом сообщать. Они не висят в кэше, они пишутся на диск ровно с целью того, чтобы данные не были потеряны.

Вопрос из зала: Просто в дополнение предыдущему. Дедупликация, на самом деле, очень замечательная вещь. Она, конечно, стоит денег, но...

Илья Космодемьянский: Вообще вещь полезная.

Вопрос из зала: Если я хочу чтобы файл сохранился, я пишу его в два места. Если под этим дедупликация — он пишется в одно место. Это меня обманывает. В результате — смысл

в этом? Я понимаю, что дедупликация — хорошая вещь, когда я сжимаю большие объемы данных. Но я говорю про другое, что не надо дедуплицировать, не надо вообще это бэкапить.

Вопрос из зала: Я имею в виду, что пишите-то в два места, но если вы 15 Postgres серверов скачиваете в одно хранилище, то пусть они занимают поменьше его.

Вопрос из зала: Может, их вообще не надо скачивать?

Илья Космодемьянский: На самом деле, тут очень важный момент, что все-таки процедура Recovery должна быть достаточно простой. С одной стороны, если мы можем жить с дедупликацией и четко понимать, где у нас в какой момент времени что есть, ее, конечно, разумно использовать, потому что иначе возникают очень большие объемы. Но если мы из-за этого рискуем процедурой Recovery или нам нужна очень высокая квалификация специалиста, чтобы ее произвести, то, может быть, лучше пожертвовать местом в пользу простоты.

Вопрос из зала: Илья, спасибо за доклад. Говорили про тестовые Recovery регулярные. Может быть, их как-то автоматизировать пробовали или даже тестирование бэкапов регулярное автоматическое?

Илья Космодемьянский: Автоматизировать-то можно все, но все равно на них лучше посмотреть глазами. Если, действительно, это процедура, происходящая раз в неделю, то не поленитесь, посмотрите глазами, почитайте логи. Но если вы хотите автоматизировать это дело, подключите — не знаю — Nagius, чтобы он сигнализировал о том, что ошибки происходят. Больше автоматизации там не придумаешь, честно говоря.

Вопрос из зала: Спасибо большое за доклад. Можете сказать пару слов про систему бэкапирования, типа Bacula или подобные?

Илья Космодемьянский: Из неинтерпрайзных бэкапов я, в основном, работал только с Amanda'ой и с Bacula'ой. Про Amanda'у я бы осторожно высказывался, с ней разных косяков было много. Bacula — вещь, в принципе, хорошая. Мне она нравится даже больше, чем HP'шный Data Protector, потому что гораздо прямее сделана.

В принципе, это обертка вокруг тех самых низкоуровневых Unix'овых утилит, про которые я говорил. Только там нормальная распределенная система, поддерживающая как виндовые, так и Linux'овые клиенты. Что немаловажно, каталог бэкапа живет в базе. Там можно либо MySQL, либо SQL Light и, по-моему, Postgres тоже можно использовать. Это надежнее, можно не продублировать свои бэкапы. С каталогом работать хорошо. Bacula я могу рекомендовать для использования.

Вопрос из зала: Вы говорите, что храните инкрементные бэкапы. Инкременты вы чем делаете?

Илья Космодемьянский: Смотря какая база или смотря какие утилиты используем. Самый простой способ — если мы используем cpio. Каждый Unix'овый файл, который там пишется, имеет 3 временных атрибута. Atime (Access time), Ctime (Create time) и какой там еще третий?

Вопрос из зала: Modification time.

Илья Космодемьянский: Да. Соответственно, когда мы создаем первый cpio бэкап level 0, мы пишем файлик о том, что он создан. Просто в директории, которую мы бэкапим. Дальше, когда делаем следующий бэкап, мы find'ом ищем файлы, которые новее чем тот файлик, который мы создали при нулевом бэкапе. Это самый простой наколенный способ создать эти инкременты — проверять по изменению Atime, а, грубо говоря, find будет писать новые вещи.

Если мы делаем базу. Если это Oracle, там в RMAN'е говоришь: «backup database incremental level 1» — и дальше все поехало.

Вопрос из зала: То есть вы инкременты пофайлово делаете?

Илья Космодемьянский: Если я бэкаплю файлы, то по файлам. Если базу, то все-таки предпочитаю, чтобы это делала специальная система в базе. В Postgres ее нет. В Percona'вском бэкапе вроде level анонсированный, насколько я понимаю. Никогда им, честно говоря, не пользовался. В Oracle, в DB2, в SQL сервере это все работает, конечно же.

Вопрос из зала: А на уровне файловых систем, если снэпшоты и ZFS?

Илья Космодемьянский: Насчет ZFS не скажу. Я рекомендовал бы очень аккуратно к ней относиться. Для бэкапа я бы ее не использовал. Там столько пряников, например, с настройкой кэша, что в реале вы получите бэкап, который будет реальной HighLoad задачей. Не искал бы я вот этих проблем на ровном месте. Это мое сугубое мнение. Говорят, что есть люди, которые умеют дрессировать ZFS.

Вопрос из зала: Полгода назад на РИТе был доклад про распределенное файловое хранилище. Там как раз на ZFS снэпшотах и инкрементах переносились данные.

Илья Космодемьянский: Да, я это знаю. Но я бы вместо такой архитектуры рекомендовал синкать файлы либо tar'ом с минус g опцией (если я не путаю опцию), либо rsync'ом все-таки. Это дубово, надежно, прозрачно и гораздо лучше работает.

Вопрос из зала: Спасибо.

Вопрос из зала: Очень хороший был доклад, очень правильные слова были сказаны. В продакшне с этим сталкивались многоократно — что-то там недотестировали, недосмотрели, например. Надо еще обращать внимание не только на тестирование самих бэкапов, на восстановление, но и на тестирование системы бэкапа. Однажды в целях улучшения бэкапа в Oracle включили компрессию бэкапа, и внезапно обнаружилось, что компрессия требует ресурсов CPU. Они внезапно кончились и завалили онлайн-базу продакшну большим количеством клиентов.

Илья Космодемьянский: Да, это очень поучительная история. Я совершенно согласен.

Вопрос из зала: Спасибо большое.

Илья Космодемьянский: Более того, я могу дополнить эту историю совершенно замечательной штукой про Data Protector. HP Data Protector умеет забирать из некой директории сделанные файлы, которые туда положили. Есть два страшных ограничения. Если у вас в Oracle используется flash_recovery_area, которая как раздел регулярно пишется, то в Data Protector'е половина этих бэкапов, скорее всего, будут битыми. Когда он их забирает оттуда своим агентом (если не напрямую Oracle пишет, не использует его просто как драйвер для ленты), то они будут битыми.

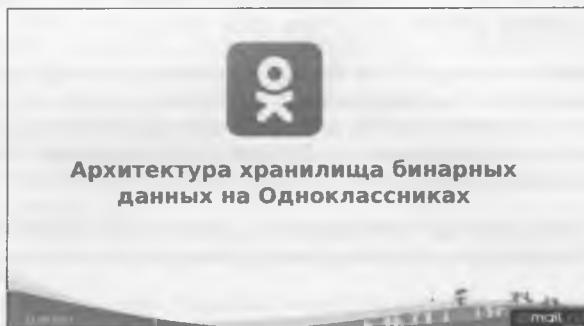
Если у вас используется директория, которая выше имеет symlink, и Data Protector туда пишет бэкапы, они тоже будут битые. Вообще эту штуку надо тестировать и внимательно смотреть в логи обоих систем (и бэкапной, и баз данных), чтобы понять вообще что у нас происходит. Так что это действительно всегда очень важно.

Вопрос из зала: Спасибо вам огромное!

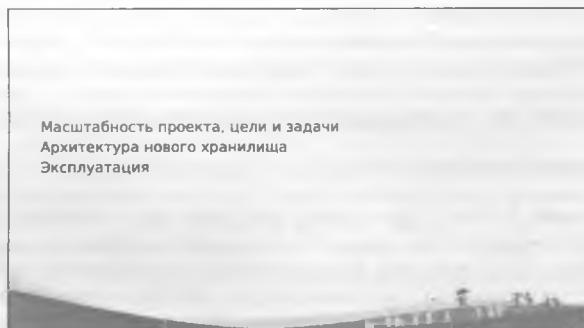
Илья Космодемьянский: Спасибо!

Архитектура хранилища бинарных данных на Одноклассниках

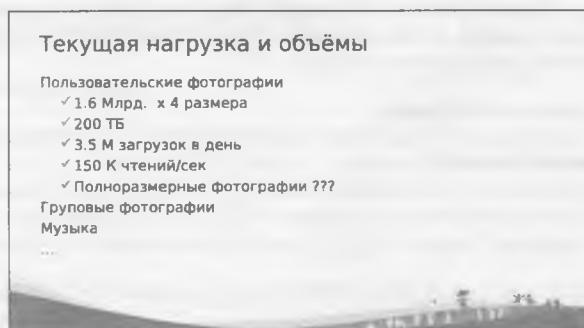
Александр Христофоров, Олег Анастасьев



Александр Христофоров: Добрый день всем! Меня зовут Александр Христофоров. Я представляю компанию «Одноклассники». Очень рад всех вас видеть. Очень надеюсь, что что-то полезное в моем докладе для вас будет. Мы долго думали, про что рассказывать. У нас много чего интересного появилось в последнее время (внутри, я имею в виду). Решили, что рассказ про новое хранилище для бинарных данных будет наиболее интересен для вас.



Я разбил доклад на 3 части. Сначала расскажу, какую задачу мы решали и почему мы ее решали. Потом расскажу, как мы решали эту задачу. В конце — какие результаты мы получили.



Текущая ситуация на портале у нас такая. Мы имеем где-то 1,6 миллиарда пользовательских фотографий. Это самое основное, о чем мы говорим, когда подразумеваем бинарные данные. Мы храним их в четырех размерах, поэтому получается 6,4 миллиарда.

Все бы ничего, но каждый день у нас загружается еще 3,5 миллиона фотографий. Это больше чем миллиард за год. Растем достаточно быстро. Сейчас мы подбираемся уже к 200-м тера-байтам данных.

В отличие от фотохостингов у нас еще эти фотографии смотрят довольно активно. (Смех, аплодисменты в зале). Мы имеем где-то 150 тысяч чтений в секунду (это на все размеры, суммарно). Мы очень хотим дать возможность нашим пользователям загружать полноценные, полноразмерные фотографии — либо оригиналы, либо очень большие фотографии, чтобы не терять качество.

Кроме того, у нас еще есть групповые фотографии. Есть музыка, видео, смайлики, подарочки и много другого интересного контента.



До недавнего времени у нас было решение, которое нас худо-бедно устраивало. У нас был BerkeleyDB, к нему некий Remote Interface. Мы использовали master-slave репликацию. Remote Interface также позволял нам делать партиционирование. Этого хозяйства у нас достаточно много, чтобы обслуживать текущую нагрузку.

Он нас не очень устраивает, потому что, прямо скажем, производительность так себе. Плохо с отказоустойчивостью. При вылете slave'a может заткнуться master, все плохо. Если потеряли master — тоже все плохо. Понятное дело.

Обслуживать это добро тоже очень тяжело. Во-первых, масштабировать можно только ровно в два раза. Во-вторых, мы не можем потушить посреди дня сервер. Нам нужны ночные маневры с down time. Это куча ручной работы, если мы хотим распартиционировать данные.

Кроме того, все это лежит на достаточно дорогом оборудовании. Мы используем внешние дисковые массивы. Мы уже просто устали покупать это железо, хотим спрыгнуть на что-нибудь более дешевое.

Также мы делали бэкапы, потому что BerkeleyDB может попортить данные. Мы делали копии. У нас хранится две копии бэкапа. Мы посчитали, что суммарно мы сейчас храним 6 копий — на master'e 2, потому что write, на slave'e 2, потому что write, и 2 бэкапа. В последнее время ситуация стала совсем печальной, потому что бэкап стал идти 17 часов. Мы просто могли не успеть за день забэкапить.

Цели

Максимальная производительность на чтение
 Отсутствие SPOF
 Резервирование вместо бэкапов
 Быстрое и гибкое расширение кластера
 Дешёвое железо
 JAVA

Мы собирались с ребятами в мае, решили, что за лето надо срочно как-то решать. Нарисовали себе цели, которые мы хотим достигнуть с какой-то новой технологией.

- В первую очередь, нас интересовала максимальная производительность на чтение.
- Во-вторых, мы хотим иметь возможность изъять любой компонент из системы, при этом все должно работать.
- Уйти от схемы бэкапов, перейти на резервирование. Хранить 3-4 реплики фотографии, чтобы при потере одной из них можно было восстановить ее обратно из двух оставшихся. При этом чтобы все работало.
- Кроме того, администраторы очень настоятельно просили нас, чтобы добавление железа было более простым, чтобы можно было сделать в полуавтоматическом режиме.
- Как я говорил, очень хотели перейти на обычное железо. Обычные дешевые сервера с обычными дисками, без всяких этих штучек.
- Кроме того, весь портал у нас написан на Java, поэтому мы очень хотели, чтобы это решение было на Java. Мы знаем, что везде, во всех решениях есть баги. Если это Java, то мы можем легко его сопровождать, поддерживать, допиливать и все остальное.

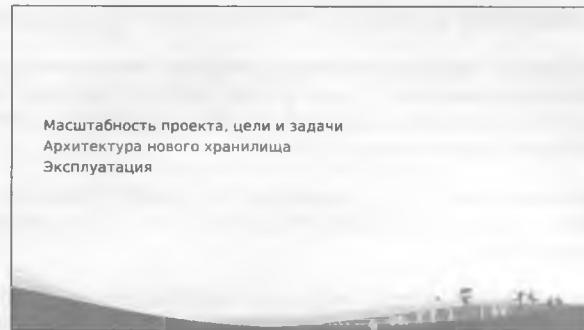
На что мы смотрели

- ✓ Распределенные файловые системы
- ✓ HDFS
- ✗ Cassandra, Voldemort, Krati

Мы посмотрели, конечно же, на что-то готовое. Думали: «Может, что-то возьмем, подойдет». Посмотрели на разные файловые системы распределенные. Очень плотно смотрели на HDFS, особенно с точки зрения хранения больших данных, типа видео или еще чего-нибудь.

Также мы смотрели на современные NoSQL-базы. Даже больше с той точки зрения, чтобы почертнуть какие-то идеи, не столько, сколько их взять.

В конечном итоге мы поняли, что решение, которое подошло бы для маленьких фотографий, которые весят 5 килобайт, и для видео (потенциально), которое весит 100 мегабайт, нет ничего готового. Мы подумали, что проще самим написать.



Дальше я расскажу, что мы написали, что у нас получилось.



На самом деле, все достаточно просто. У нас есть некие сервера, которые обслуживают диски. Они работают с дисками. Эти сервера предоставляют некий Remote Interface наружу, через TCP и через HTTP. Для больших данных это может быть полезно.

Кроме того, у нас есть кластер Zookeeper, который мы используем для координации работы всех этих серверов. У нас есть клиентская библиотека, которая работает с Zookeeper'ом, она знает о состоянии кластера и потом ходит к нодам за данными либо кладет туда данные.

Наши типичные клиенты — это наши HTTP сервера. Если интересно, это TomCat'ы. Для маленьких фотографий есть большой кэш внутри, для больших фотографий, соответственно, его нет. В принципе, мы в будущем рассматриваем возможность отдавать большой контент непосредственно с нод, на которых данные лежат, без промежуточного HTTP сервера. Он будет только формировать линк.



Теперь я попытаюсь поподробнее рассказать, что такое этот сервер-хранилище.

Первое решение, которое мы приняли (я считаю, что оно все-таки было правильным). Мы работаем с каждым диском независимо. У нас нет как такового сервера, мы работаем именно с дисками. Плюс в том, что можно диск вытащить из ноды, поставить другую ноду, и все будет работать дальше. Абсолютно без разницы, где он стоит.

Другой плюс. Если диск вылетает, он не оказывает абсолютно никакого влияния на другие диски и работоспособность сервера целиком. Риск того, что что-то заткнется из-за того, что вылетел один диск, минимален.

Как же мы все это дело храним. Файлов у нас много, особенно мелких фотографий. Они сами по себе по размеру маленькие, но их очень много. Понятно, что хранить каждую фотографию в своем файле, это очень дорого. Поэтому мы храним их в больших сегментах. Это здоровые файлы, в которых помещается куча таких фотографий или куча файликов маленьких логических.

Плюс у нас есть индекс в памяти. Индекс мы храним целиком в памяти, чтобы абсолютно минимизировать количество sick'ов по диску. Когда мы записываем данные, мы просто берем сегмент, записываем туда кусок данных, запоминаем, куда мы его записали, пишем эту информацию в индекс. В индексе еще есть transaction log, про который я поподробнее расскажу потом.

Наружу этот сервер предоставляет интерфейс по TCP, как я говорил. Мы используем Apache и Mina, это неблокирующий сервер.



Сегменты данных. Они могут быть любого размера, но они должны быть все одинаковые. Сейчас мы используем сегменты размером 256 мегабайт. Когда сегмент создается, мы сразу же под него резервируем место с помощью xfs_io. Для чего это надо — для того, чтобы избежать фрагментации. У нас нет файлов разных размеров, поэтому мы всегда знаем, что файл — это один непрерывный кусок диска.

Также у нас есть только один активный сегмент на диск, через который идет запись. Мы пишем всегда только в один файл. Но опять же таким образом мы пытаемся минимизировать количество sick'ов и влияние записи на чтение.

Дальше мы решили, что мы не будем бороться с удалениями посредством запоминания пустых мест файлов и всего остального, потому что это и память, и сложность кода. Это никому не надо. Мы просто уплотняем сегменты со временем. Когда набирается 30% свободного места в сегменте, мы берем из него данные, пишем в новый активный сегмент, старый помечаем как свободный. В него могут потом начать писать данные.

Индекс

Хеш таблица на десятки миллионов элементов

- ✓ увеличение размера без паузы
- ✓ 1 примитивный массив + direct memory
- ✓ данные в памяти = данные на диске

Лог транзакций

- ✓ Изменения индекса → логи на отдельном диске
- ✓ Синхронизация и чистка логов
- ✓ Проверка целостности данных при старте

Что такое индекс. По сути, это хеш-таблица, в которой по ключу лежит адрес. Адрес — это номер сегмента и смещение в сегменте. Но так как все на Java, то обычный hash map взять нельзя, будут проблемы с garbage collector'ом, потому что там десятки миллионов элементов. Для маленьких фотографий суммарно на сервере больше миллиарда фотографий.

Плюс у hash map'a есть другая особенность. Когда он заполняется, его нужно растянуть в два раза. Это пауза и обработка встает. Поэтому там просто написан свой map, который умеет растягиваться без паузы. Он построен на одном массиве int'ов. Данные реально лежат в direct memory. Таким образом, с точки зрения garbage collector'a один hash map — это два объекта.

Кроме того, данные в памяти лежат абсолютно точно так же, как на диске, один в один. Это позволяет нам очень быстро поднимать сервер, тушить его, очень быстро синкать сервер с диском. Не нужна никакая трансформация данных. Взяли кусок байтов, записали — и все.

Сам индекс в памяти, но потерять его не хочется. Поэтому у нас есть transaction log, в который мы пишем все изменения. Он выглядит абсолютно точно так же, как и сам индексный файл. Периодически мы просто берем существующие transaction log'и... Вернее, мы берем изменения, которые есть в памяти на текущий момент, мы их flash'им на диск в индексный файл и удаляем логи. Мы считаем, что мы эти логи обработали.

Когда сервер поднимается, он смотрит: если есть какие-то логи, он их накатывает, потом делает тот же самый снапшот и чистит их. Кроме того, когда сервер поднимается и в логе находит какие-то новые данные, мы еще проверяем сегментные файлы на всякий пожарный на предмет их целостности. Там хранится CRC, мы читаем данные, смотрим — CRC валидно, валидны данные. Если не валидны, мы говорим, что лучше их восстановить, потому что, похоже, мы их недописали до конца.

Кластер и Резервирование

Уникальный ИД диска

Фактор репликации - 3

Равномерное распределение реплик

Нет 2 реплик на одном сервере

Кворум записи — 2

Чтение — 1 + 1

Теперь немного расскажу, как это все вместе функционирует.

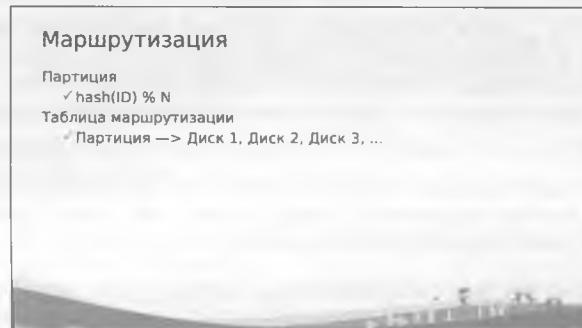
У каждого диска есть уникальный ID. Он заводится, когда диск активируется в системе. Он хранится в файликсе.

Фактор репликации, который мы сейчас используем для фотографий, это 3. Мы всегда имеем 3 реплики данных.

Реплики мы размазываем достаточно равномерно по кластеру. Нет такого, что данные одного диска среплицированы на двух других дисках. Нет, они будут размазаны между 20-ю дисками.

Единственное правило — чтобы двух реплик не было на одном сервере. Понятно для чего. При записи мы используем кворум. Мы считаем, что запись была удачной, если мы смогли записать на две ноды. При чтении же, если мы записали хотя бы с одной ноды, мы считаем, что нам удалось.

Но сейчас, если данных нет по ID, то мы сходим еще на один сервер и проверим, может быть, они там есть. Это, скорее, временная мера. Может, не времененная. В очень редких случаях может произойти такое, что на одном сервере она есть, до второго еще репликация не доехала, например. Не репликация — я расскажу поподробнее.



Как мы понимаем, где какие данные лежат. Серверов много, дисков много. Сначала мы отталкиваемся от некоего понятия «партиция». Это логическая штука. Например, мы разбили все данные по текущей конфигурации на 5 тысяч партиций. Для каждой партиции мы просто потом храним остаток от деления, hash отключен.

Для каждой партиции мы храним буквально 3 номера диска, на которых лежат все реплики. Это мы называем таблицей маршрутизации. Для каждой партиции в явном виде прописано. Это очень удобно, потому что это работает очень быстро, с одной стороны. С другой стороны, вся логика построения этой таблицы маршрутизации вынесена, она вне системы. Это какой-то внешний tool, который может посчитать. Вы можете создать таблицу маршрутизации руками, туда зааплодить, и все у вас полетит.



Как же мы расширяем это дело. Когда мы хотим добавить некие новые диски, то мы просто-напросто создаем новую версию таблицы маршрутизации.

Пример такой. У нас была одна ситуация. Было двеパーティション, однаパーティション лежала на диске 1, 2, 3, вторая — на диске 4, 5, 6. У нас есть какой-то tool. Мы говорим ему: «Теперь мы хотим, чтобы у дисков 1, 2, 3 был вес 0. Мы на них больше не пишем. Зато у нас появились 7, 8, 9, мы на них точно так же хотим писать». Он возьмет этот инструмент, каким-то образом посчитает, скажет: «Все понятно, теперь здесь 7, 8, 9. Этаパーティション ложится теперь на эти диски».

Таким образом, у нас есть вся история изменений кластера. Мы называем их «регионы» (я называю их «регион»). Для каждого региона мы знаем, грубо говоря, список ID, которые должны туда попасть. В нашем случае это очень просто. Это начальный ID и конечный ID. Если ID попал в этот регион, значит, мы берем эту таблицу. Если ID попал в этот регион, значит, мы берем эту таблицу маршрутизации. Все очень просто.

Преимущества Регионов

Расширение не требует передвижения данных
Не надо хранить местоположения для каждого объекта
Каждый клиент имеет реплику всех версий таблицы
маршрутизации в памяти

Что нам это дало.

С одной стороны, мы можем добавлять железо, убавлять железо. При этом нам не надо физически двигать данные. Я считаю, это большой бонус. Все-таки это занимает время, это довольно сложно. Мы всегда можем это сделать. Но в нашем случае это не является обязательным.

С другой стороны, есть системы, которые позволяют не двигать данные, зато (например, как HDFS) они хранят для каждого файла его местоположение на диске. Для каждой фотки нам бы пришлось хранить местоположение. Это лишняя инфраструктура, лишний поход (latency), сложности и все остальное.

А так у нас все очень просто. У нас каждый клиент имеет реплику таблицы маршрутизации всех версий. Она достаточно маленькая, чтобы поместить ее в память каждого сервера.

Zookeeper Для Координации

Что такое Zookeeper?
Сервис для координации распределенных приложений
Дерево + Бинарные данные в узлах

Особенности Zookeeper:
• Мастер-сервер
• Помехи к легким
• Кворум записки
• Гарантированная последовательность
• Установка предпочтений
• Конфигурации

Почему Zookeeper?
Надежный >= 3 серверов
Быстрый

Немного расскажу про Zookeeper, как мы его используем. В принципе, Zookeeper себя позиционирует как сервис для координации распределенных приложений. По большому счету, это некая база данных. Дерево, в узлах которого могут быть произвольные бинарные данные, массив байтов. Это такая распределенная система, все хранится в памяти. Изменения пишутся в transaction log. Для записи точно так же требуется кворум, как и у нас.

Интересная особенность его в том, что изменения считываются клиентом ровно в той же последовательности, в которой они были сделаны. Это очень важно. Эта гарантированная последовательность дает возможность делать всякие хитрые штуки с ней.

Еще интересная штука — это временные вершины. Это вершина в дереве, которая автоматически удаляется, когда Zookeeper потеряет соединение с клиентом. По факту это не просто соединение с клиентом. Это соединение с кластером, грубо говоря. Это таймаут сессии.

Плюс полезная штучка — нотификация. Можно подписаться на изменения узлов в дереве и, таким образом, сильно оптимизировать походы к серверам.

В общем, мы решили, что для нас он достаточно надежный. Мы можем поставить там 3 и более серверов. Он достаточно быстрый для нашей задачи. Поэтому мы решили его применить как раз для координации.



Что мы там храним. Мы храним, в первую очередь, сервера и их IP-адреса. Также мы храним диски и их статусы. Когда сервер поднимается, он говорит: «Слушай, у меня есть такой-то диск, такой диск, он сейчас в статусе таком-то. Я сам такой-то, у меня IP-адрес такой-то, я сейчас живой».

Таким образом, конфигурация серверов очень простая. Все, что нам нужно знать, это адреса Zookeeper серверов и какое-то семантическое имя кластера наше. Мы реально там храним для нескольких — пяти-шести — кластеров, для всех будем хранить конфигурацию.

Кроме того, мы там же храним таблицы маршрутизации, все их версии. Также мы используем Zookeeper для того, чтобы организовать некую распределенную блокировку. По факту она нужна только в момент, когда меняется маршрутизация. Когда мы добавляем новую версию, чтобы гарантировать, что никто не зааплоадил новую фотографию по старой, неправильной версии таблицы маршрутизации, мы вынуждены сделать блокировку. Это происходит, наверное, в течение секунды.

Обработка Ошибок

Тип проблемы	Обнаружение	Обработка
Вылет диска	IOException	Выключение диска
Вылет сервера	Таймаут сессии в Zookeeper	Вывод из кластера
Потеря соединения к серверу	Ошибки выполнения запроса	Блокировка сервера и мониторинг

Надежность — это было одним из самых важных, о чем мы заботились. Надежность подразумевает под собой борьбу с проблемами и правильную реакцию на них.

Типичная проблема — это вылет диска. Понять несложно — это ошибка ввода-вывода. Мы просто в кластере помечаем этот диск как «мертвый». Все об этом достаточно быстро узнают. На него просто перестают идти запись и чтение.

Если вылетает какой-то сервер, то тут нам помогает Zookeeper. Сессия просто-напросто таймаутится, нода пропадает из дерева, и все быстренько тоже узнают.

Но есть, во-первых, какой-то период времени, пока все узнают об этом. Во-вторых, просто клиент может потерять соединение до сервера. Сетевая проблема. Именно этот клиент этого сервера временно потерял соединение.

В этом случае мы получим, скорее всего, какие-то ошибки выполнения. Клиент просто заблокирует сервер на время и начнет его мониторить на доступность. Как только он появится обратно, он начнет его опять вызывать. При этом из кластера этот сервер не выводится.

Hinted Handoff и восстановление

Hinted Handoff

- ✓ Недоступность 1 реплики → 2 реплики хинта
- ✓ Чтение хинтов при старте, диск доступен на запись
- ✓ Периодическая проверка хинтов

Полное восстановление данных

- ✓ Диск доступен на запись
- ✓ Восстановление параллельно с разных дисков

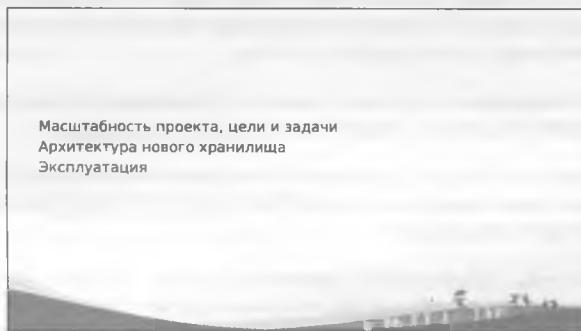
Что же делать, когда кто-то все-таки был недоступен. В нашем случае, если мы записали на две ноды, мы считаем, что все удалось. Одна реплика была недоступна, например. Тогда мы пишем хинты. Хинт — это, по сути, ключ, тип операции (удаление или добавление) и время. Пишем мы точно так же — две реплики для надежности этих хинтов. Записать мы их можем на абсолютно любые два произвольных сервера.

Когда сервер поднимается, он пытается прочитать хинты для всех дисков, которые у него есть. Только тогда, когда он их прочитал, диск становится доступен на чтение. До этого диск доступен только на запись.

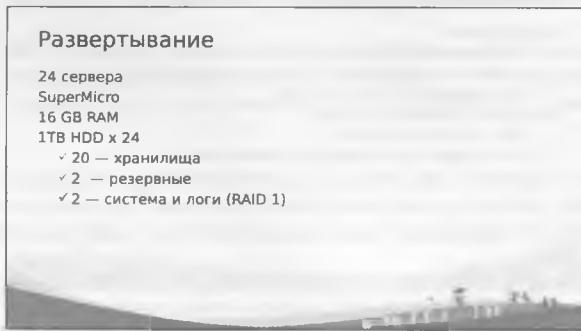
Плюс мы периодически проверяем хинты в процессе работы (каждые 30 секунд, грубо говоря). Это опять же на случай того, что конкретный клиент до конкретного сервера потерял соединение, например, он не смог записать конкретную фотографию. Тот сервер все равно получит ее. Через 30 секунд он посмотрит, что где-то есть хинты или соберет эти данные со своих соседей. По факту у нас была ситуация, когда что ни день — эти хинты... Сервер апдейтился только через хинты, и все работало.

Вторая типичная проблема — это не кратковременное падение, а полный вылет диска, когда нужно данные восстановить. Мы просто втыкаем диск, говорим «восстановить». Диск автоматически становится доступным на запись, но недоступным на чтение, что логично.

Так как я говорил, что у нас реплики размазаны достаточно равномерно, то он параллельно с кучи дисков начинает восстанавливать данные. Это будет порядка 20 дисков, например. Восстановление происходит очень быстро. Главное, что оно не оказывается никаким образом на функционировании остальной системы.



Теперь я попытаюсь рассказать, где мы это запустили.



Первое, где мы запустили. Мы запустили на больших пользовательских фотографиях. У нас их 1,6 миллиарда. Они где-то в среднем по 60 килобайт. Итого мы собрали кластер из 24-х серверов. SuperMicro — дешевые сервера по 16 гигабайт памяти, по 24 диска в каждом.

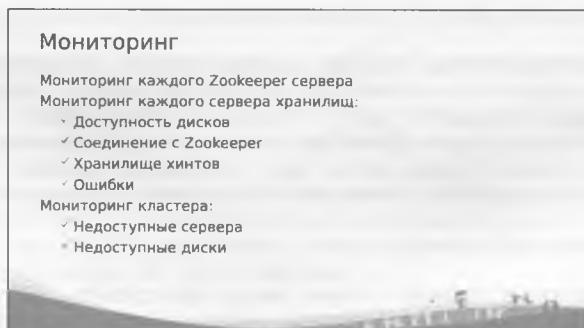
Из них мы 20 реально используем под данные, 2 диска под систему и под transaction log'и (они в RAID'e), 2 у нас резервные. Почему 2 резервные. Мы посоветовались с администраторами, они нам сказали, что это будет самая правильная и логичная схема. Если вылетает диск, то можно сразу пойти, кнопочка «восстановить данные». Они будут восстанавливать на этом же сервере, но на соседнем диске. Потом со временем, когда будет время, кто-то придет и поменяет.



Как мы оценивали результаты работы и мерили. У нас есть очень много всякой статистики. Есть сбор статистики серверов, CPU, сеть, все остальное. Плюс у нас есть очень много всякой статистики из приложения. Мы можем посмотреть среднее время, ошибки, количество вызовов, максимальное время. Все это мы можем сделать по конкретному серверу, по клиенту, по методу, по диску, по комбинации клиент-сервер. В общем, куча всякого добра.

Заданный latency... Мы провели сначала синтетическое тестирование с заведомо большей нагрузкой, чем у нас будет в продакшне. Мы получили порядка двух тысяч чтений в секунду с сервера (это 100 чтений с диска). По большому счету, там больше ничего не выжмешь, потому что это rotational диск.

Понятно, что если фотографии большие, то там с другой стороны начинается упор в сетку. 2 тысячи на 60 килобайт — явно больше, чем гигабит. У нас стоят сейчас гигабитные интерфейсы пока. Максимум мы можем 900 мегабит. Нам этого хватает выше крыши, потому что наша номинальная нагрузка сейчас, под которой они в продакшне, это где-то 600 чтений в секунду и 350 мегабит в секунду. То есть у нас есть в 2,5 раза запас (где-то так). Пока все счастливы.



Немного про мониторинг. У нас есть внутренняя система мониторинга, с помощью которой мы мониторим, во-первых, каждый Zookeeper сервер, во вторых, каждый сервер хранилищ, в том числе доступность дисков, не потерялось ли соединение с Zookeeper'ом, хранилище хинтов (для Hinted Handoff'a) и в норме ли ошибки.

Плюс мы мониторим весь кластер целиком, проверяем, может, кто-то потерялся просто. Мы считаем, что должен быть какой-то диск, а его просто нигде нет в системе.



Вот и все. Меня зовут Александр Христофоров. Мой коллега Олег Анастасьев. Мы оба работаем с «Одноклассниках». Если есть какие-то вопросы, я, конечно, с удовольствием сейчас отвечу и там отвечу. Можете найти нас на «Одноклассниках» по таким нехитрым линкам.

Спасибо.

Вопросы и Ответы

Вопрос из зала: Из вашего доклада не совсем понял. На диске, который поднимается, есть какая-то файловая система или вы просто бьете его, грубо говоря, низкоуровневыми операциями?

Александр Христофоров: Мы используем xfs.

Вопрос из зала: Здравствуйте! Спасибо. Скажите, пожалуйста, как происходит удаление файла? Я правильно понял, что когда происходит закачка, она происходит с 20-ти размазанных дисков, вы говорите, но это все равно физически одна нода. Или с нескольких нод происходит закачка?

Александр Христофоров: Нет-нет. Это будут разные ноды. Когда мы строим таблицу маршрутизации, реплики пытаются размазаться достаточно равномерно. Они попадают... Скажем так, сейчас на один диск приходится где-то 10 партиций. Мы выбрали такую конфигурацию логически. Эти 10 партиций, скорее всего, будут более или менее равномерно размазаны по 20-ти дискам. Эти 20 дисков, скорее всего, будут на разных машинах. Это вероятностная штука, поэтому мы не следим, чтобы это было так уж строго соблюдено.

Вопрос из зала: Понятно. Удаление?

Александр Христофоров: Удаление — точно так же, как апдейты. Мы пишем в индекс, что все, больше нет данных. В файле остается дырка, мы помечаем, что реально в файле используется не 256 мегабайт уже, а 255, чтобы потом определить, когда нужно сделать compaction. Все. Там точно так же — если есть какие-то проблемы, то есть хинты, они точно так же пишутся, потом обрабатываются, если сервер был дан.

Вопрос из зала: Спасибо.

Вопрос из зала: Скажите, пожалуйста. У вас восстановление происходит путем перезаписи полностью одного диска.

Александр Христофоров: Да.

Вопрос из зала: Сколько это примерно занимает времени?

Александр Христофоров: Сложно сказать.

Вопрос из зала: 100 мегабайт в секунду в лучшем случае на...

Александр Христофоров: Это часы. В пределах дня.

Вопрос из зала: В вашей вероятностной модели выход из строя еще двух дисков с какой вероятностью происходит за это время?

Александр Христофоров: Когда мы только запустили, когда железо было все новое, необкатанное, у нас вылетела нода и два диска. В принципе, все работало. Единственное, что какой-то маленький процент людей не мог закачивать фотографии. Это понятно, потому что пересечение есть при таком количестве. По сути, мы потеряли 26 дисков в тот момент.

Вопрос из зала: Нода и два диска на разных нодах?

Александр Христофоров: По-моему, да. Два диска на разных нодах и ноду целиком.

Вопрос из зала: Если равномерно размазывать, то получается, что при выходе из строя трех дисков у вас практически со стопроцентной вероятностью должны теряться какие-то файлы, если они навсегда выходят.

Александр Христофоров: В принципе, да. Если даже так совпадет, и мы потеряем 3 диска, и там действительно будут все 3 реплики этих данных (что крайне маловероятно), то мы потеряем очень маленький процент данных. Это лучше, чем иметь полную реплику на трех дисках и вдруг (с еще меньшей вероятностью) потерять 3 реплики. Тогда будет совсем все плохо. Или хотя бы две из трех.

Вопрос из зала: Не рассматривали ли возможность восстановления, например, со всех дисков на все, чтобы снизить время из-за того, что партиционирование было бы намного сложнее?

Александр Христофоров: Ну, да. Честно говоря, я даже не думал над этим.

Вопрос из зала: Контрольные суммы всегда считаете? Как еще целостность проверяете?

Александр Христофоров: Когда мы пишем, мы всегда считаем контрольную сумму. При чтении мы не проверяем.

Вопрос из зала: Контрольную сумму на каждый файл?

Александр Христофоров: На каждый файл логически. На каждую фотку в конце дописывается CRC. Есть идея запускать периодически сканы данных на предмет целостности. Можно их пробовать проверять каждый раз. Конечно, это тоже не проблема. Но это все еще пока на стадии обсуждения. Возможно, мы так и сделаем.

Вопрос из зала: Хорошо. Спасибо.

Вопрос из зала: Скажите, пожалуйста, сколько Zookeeper серверов вы используете здесь?

Александр Христофоров: Пока у нас 3.

Вопрос из зала: Они одновременно работают и отвечают, у них кластер между собой?

Александр Христофоров: Да, да. Они выбирают одного master'a, который координирует запись, а чтения идут...

Вопрос из зала: Это помимо 24-х серверов или 3 из этих 24-х?

Александр Христофоров: Нет-нет. Это помимо. Они абсолютно отдельно. Мы их поставили и планируем использовать для других задач в будущем.

Вопрос из зала: Спасибо.

Вопрос из зала: Спасибо за доклад. Вами была озвучена идея, что хранить только самое большое изображение в случае потери более мелких (их можно восстановить всегда). Но я пропустил — не реализовали эту идею?

Александр Христофоров: Я такого не говорил. Вообще это вполне разумная идея.

Вопрос из зала: Сейчас все-таки 4 версии картинки хранятся?

Александр Христофоров: Сейчас мы храним все 4, да. Наверное, это имеет смысл. Может быть. Пока мы решали конкретную задачу — взять и переложить данные из старого хранилища, которое уже дышало на ладан, в новое. Мы не пытаемся решать сверхзадачи.

Вопрос из зала: Вы говорили, что вы фотографии группируете и складываете в больших файликах. Не пробовали хранить их просто на Razer'e — каждая фотография в своем файлике?

Александр Христофоров: Нет, не пробовали.

Вопрос из зала: Смысл был — реализовывать свою систему директорий, а когда такие системы, как Razer FS вполне справляются с десятками миллионов файлов вполне хорошо.

Олег Анастасьев: Я отвечу. Мы как-то использовали Razer FS с точки зрения быстродействия как раз на работе с кучей маленьких файликов. Он сожрал весь CPU, не смог даже один диск обработать.

Вопрос из зала: Там, может быть, группировать каталоги можно было?

Александр Христофоров: Мы и так, и сяк группировали.

Вопрос из зала: Просто у меня получалось 100 миллионов файлов хранить на одном диске. Единственное, что mountain mount происходит. Это был первый вопрос. Второй вопрос — вопрос производительности. У вас при большой работе с диском нет ли проблем с работой с сетью? Вы же Linux используете?

Александр Христофоров: Да.

Вопрос из зала: Я заметил, что у Linux есть такая особенность, что если начинается большое число записей, то почему-то начинает тормозиться весь сетевой стек. У вас такого не наблюдалось? Если наблюдалось, как вы решали это и решили ли?

Александр Христофоров: Да, я согласен, деградация есть. При этих пиковых нагрузках, когда речь идет уже о 900 мегабитах, есть деградация ощутимая. Но опять же для нас она лежит в пределах нормы. Это все равно лучше, чем наше старое решение. Ничего плохого в этом нет. Пока, честно говоря, мы с этим особо не возились. При номинальных нагрузках таких проблем точно нет.

Вопрос из зала: Понятно. Эта проблема легко решается с помощью NFS. Но для этого надо много маленьких файликов. С большими файлами по NFS не поработаешь. Просто такое замечание.

Александр Христофоров: Опять же — 100 миллионов, миллиард — может быть, есть большая разница.

Вопрос из зала: Спасибо за доклад. Можно немного истории — какие решения у вас были до этого, на каких объемах, масштабах вы поняли, что вам приходит game over?

Александр Христофоров: Я говорил: у нас был BerkeleyDB. Олег знает, что было еще до этого.

Олег Анастасьев: В самом-самом начале в качестве бинарного хранилища использовалась SQL-база. Это работало на 2-3 тысячах пользователей, которые были сначала. Потом стало понятно, что в SQL-базе хранить невозможно. Было сделано решение, про которое Саша рассказывал, основанное на BerkeleyDB. Оно довольно неплохо работало в начале проекта. Проблемы мы стали ощущать где-то с 2009-го года.

Вопрос из зала: Скажите, пожалуйста. Вы это новое хранилище сделали для того, чтобы избавиться от проблем со старым. Как проходила миграция со старого хранилища на новое? Сколько времени это занимало? Выводили вы сразу из работы сервера или нет.

Александр Христофоров: Расскажу. Процесс был довольно непростой, небыстрый. Что мы сделали. Сначала мы собрали миникластер, нагенерили туда файловых данных, проверили, что эта схема рабочая. Что с нагрузками все хорошо, что можем вывести, завести — все отлично. Потом подняли продакшн кластер, написали штуку, которая читает из старого, пишет в новое. Параллельно уже шла запись новых данных в новое хранилище.

Эту миграцию мы запустили, она с переменной нагрузкой в течение дня, чтобы не положить существующую инфраструктуру, забирала просто одну за одной фотки. Суммарно это заняло 7 дней чистого времени. За 7 дней мы перелили все данные. Там было 3 машины, которые буквально ночью, когда нагрузка минимальная 3 гигабита писали. Грубо говоря, 1 гигабит читали, 3 гигабита писали. Таким образом, мы за 7 дней все это слили.

Потом мы переключили чтение. Посмотрели, что все читается по-новому, но запись у нас пока еще идет в 2 места. Уже месяц мы его эксплуатируем. Было страшно отключать. Пока запись идет, но на этой неделе как раз планируем отключение старого.

Вопрос из зала: Чтение уже полностью отключили?

Александр Христофоров: Да, да, чтение давно уже отключили.

Вопрос из зала: Бэкапы восстановления — запланированы эти процедуры, решается?

Александр Христофоров: Бэкапы мы сейчас не планируем. У нас есть 3 реплики. Единственное, что мы бэкапим сейчас, это индексный файл. Просто мы бэкапим его на системный диск. Он в RAID'e. Просто не для того, чтобы не потерять его из-за выпада сервера, но и из-за какой-то ошибки в коде, которая может похерить...

Вопрос из зала: Ошибка софта не страшна?

Александр Христофоров: Страшно, поэтому сделали бэкап этого индекса.

Вопрос из зала: Кроме того, что нет бэкапов, я правильно понимаю, что все 496 винчестеров находятся в одном дата-центре?

Александр Христофоров: Сейчас да. (Смех в зале).

Вопрос из зала: На самом деле, третья копия данных тогда лишняя, потому что выход из строя всего дата-центра более вероятен, чем выход из строя трех винчестеров одновременно.

Александр Христофоров: Вполне возможно.

Вопрос из зала: Здравствуйте. Скажите, если у вас поменяются размеры этих фотографий, вы как все будете делать? Перегенерите?

Александр Христофоров: Да, будем перерезать. Буквально сейчас у нас запланирована задача по... Ну, сейчас совпало очень хорошо, что мы будем их мигрировать.

Вопрос из зала: Сколько времени это может занять?

Александр Христофоров: Много, наверное. Мы не считали. Как раз посчитаем. В ближайшие недели будем запускать нарезку нового размера, и посчитаем, сколько это будет занимать.

Вопрос из зала: Понятно. Спасибо.

Вопрос из зала: Скажите, пожалуйста. Я так понимаю, вы эту систему писали с нуля?

Александр Христофоров: Да.

Вопрос из зала: Сколько человек писало это и сколько времени?

Александр Христофоров: Один человек, 3 месяца.

Вопрос из зала: Второй вопрос. Если один винчестер ломается, то откуда вы знаете, какие файлы нужно заново скопировать? Вы говорили, что вы не храните, что где лежит.

Александр Христофоров: Да. Все очень просто. Мы можем опросить все ноды. Каждая нода знает, какие у нее есть ID. Все эти ID в памяти. Обойти все ID очень дешево. Для каждой ID проверить, на каком диске все 3 реплики должны лежать, тоже очень легко. Обход всех ключей, определение, какие ключи...

Вопрос из зала: То есть вы перебираете все полтора миллиарда...

Александр Христофоров: Да. Но это очень быстро происходит, поверьте.

Вопрос из зала: Скажите, что сейчас мешает расползтись на несколько дата-центров, масштабироваться?

Александр Христофоров: (Обращается к коллеге). Хочешь, Андрюха, сказать?

Реплика из зала: Ничего не мешает, есть более приоритетные задачи.

Александр Христофоров: Вот. Ничего не мешает, есть более приоритетные задачи.

Вопрос из зала: Принципиальных...

Александр Христофоров: Принципиальных нет проблем. Мы это сделаем. Просто мы и так уже переезжали из дата-центра в дата-центр. Это не такой простой процесс.

Вопрос из зала: Здравствуйте. Вы рассматривали другие решения. Можно поподробнее рассказать какие, делали ли вы там какие-нибудь замеры? Типа в вашем решении отдачи быстрее на столько-то, диски — на столько-то дефрагментируются и так далее.

Александр Христофоров: Очень много решений мы отмели по политическим соображениям, грубо говоря, или по рекомендациям коллег.

Как я говорил, мы очень плотно тестировали HDFS, но больше для хранения больших данных, типа видео. Отмели мы его по причине того, что, во-первых, он недостаточно надежен. High availability name node не была допилена до сих пор. Плюс он очень много потребляет памяти на name нодах на поддержание этого индекса, где какой файл лежит. При нашем количестве нужно просто какие-то терабайты памяти. Это нереально. GGC бы заткнулся уже при десятой части этих файлов. Поэтому мы его отмели.

Что еще. Была еще идея взять VkRate, прикрутить к нему Remote Interface и логику репликации. Просто взять его как ту часть, которая хранит на диске. Я померил, было все не очень радужно. Там по факту приходилось больше чем 1 чтение с диска на запрос. Из-за этого он меньше мог обслужить запросов. Как-то так.

Вопрос из зала: Было ли рассмотрено решение — маленькие файлики хранить и отдавать где-нибудь в другом месте (из памяти, например), а большие, действительно, хранить как файлы.

Александр Христофоров: Когда я говорил про всю эту картинку целиком, те HTTP сервера, которые отдают картинки, на самом деле, имеют кэши внутри. Маленькие картинки (аватарки на «Одноклассниках» и мелкие картинки) практически все лежат в памяти. 150 тысяч чтений перенести на диск — я думаю, это глупо. Большие практически все попадают на диск, а мелкие — да.

Вопрос из зала: Спасибо.

Вопрос из зала: (Неразборчиво, без микрофона, 54:57).

Александр Христофоров: У нас свои все кэши написаны. Но для фотографий конкретно написан маленький кусочек на C.

Вопрос из зала: То есть это HTTP демон, он сам кэширует все?

Александр Христофоров: Это TomCat, в нем...

Вопрос из зала: Ваш модуль C'шный, который кэширует фотографии именно.

Александр Христофоров: Да, да.

Вопрос из зала: Как вы очищаете? Есть какая-то политика очистки этого кэша?

Александр Христофоров: Он вытесняет просто.

Вопрос из зала: Это не вы занимаетесь этим? Это уже готовое...

Александр Христофоров: Это тысячу лет написанная штука. Он просто вытесняет по размеру.

Вопрос из зала: Понятно.

Вопрос из зала: Скажите, почему вы храните только бинарные данные в этом хранилище, а какие-то текстовые — нет?

Александр Христофоров: Просто мы его месяц как эксплуатируем полноценно. У нас есть какие-то идеи. Возможно, мы и будем использовать его для чего-то другого. Пока только идеи.

Вопрос из зала: Сейчас текст где?

Александр Христофоров: У нас есть много чего. Есть SQL-сервера, есть по-прежнему Berkeley, для многих задач он хорошо работает. Есть Cassandra, есть Tarantool с Voldemort'ом. У нас много чего есть.

Вопрос из зала: С текстом проблем таких нет, как с бинарными данными?

Александр Христофоров: Я даже не знаю, где у нас текст. Разве что поиск — там мы храним документы.

Вопрос из зала: Все равно какие-то посты, что-то еще.

Александр Христофоров: Это все в Berkeley. Практически. Система сообщений — там тоже был Berkeley. Он был в памяти. Но сейчас мы плавно мигрируем на новое хранилище, где мы используем комбинацию Tarantool'a и Voldemort'a.

Вопрос из зала: Подскажите, какой протокол используется для передачи данных в это хранилище?

Александр Христофоров: В это — буквально бинарный протокол, очень простой, самописный.

Вопрос из зала: Тоже на каждой ноде висит какой-то демон, который реализует фактически файловую систему?

Александр Христофоров: Нет, что-то слишком сложно, не понял. Там буквально он слушает по TCP, у этого демона есть несколько команд. Команды типа «положи на такой-то диск кусок байтов», «дай мне с такого-то диска кусок байтов» и все.

Вопрос из зала: Логика связи имени файла с данными на диске (фактически сама файловая система) где реализуется?

Александр Христофоров: Мы не храним имя файла, у нас этого ничего нет. У нас есть просто ключ. Этот ключ — 8-байтовое число и тип. ID фотографии и тип фотографии — вот наш ключ.

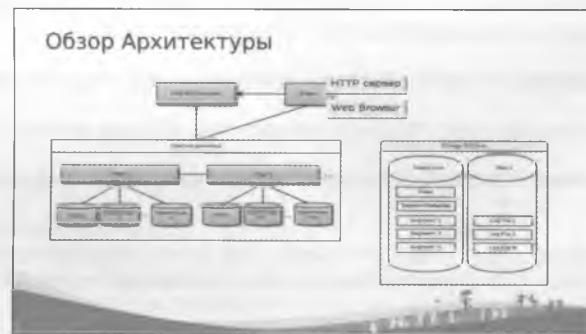
Вопрос из зала: На страницу вы что встраиваете?

Александр Христофоров: Линк. В этом линке ID, в качестве параметра ID и тип.

Вопрос из зала: Трансляцию осуществляет, таким образом, веб-сервер?

Александр Христофоров: Ну, да. Я не очень понял вопрос.

Олег Анастасьев: У нас с этой позицией более простая ситуация. У нас практически все написано на одном языке. У нас нет проблем связи разных подсистем друг с другом.



Вот этим клиентом (показывает на слайде) является веб-сервер, которыйрендерит основной портал. Он имеет у себя в памяти реплицированную копию всех версий таблиц маршрутизации. Соответственно, вопрос генерации линка для этой фотки — это просто вопрос просмотра пары таблиц в памяти и деление строчки. Это очень быстро. Это еще одна причина, почему мы хотели отказаться от какого-то центрального места, где хранится маппинг между фоткой и диском, где она лежит.

Вопрос из зала: У меня комментарий к одному из предыдущих вопросов. При перенесении этой архитектуры на несколько дата-центров будут проблемы с кворумом Zookeeper'a, он быстро работает только в одном дата-центре.

Александр Христофоров: Возможно. Честно, мы не проверяли. С другой стороны, для нас это небольшая проблема, потому что мы туда очень мало пишем по факту. Мы только читаем оттуда.

Вопрос из зала: Скажите, пожалуйста. У вас этот бинарный поток отдает TomCat. Не используете там nginx, ничего такого?

Александр Христофоров: Сейчас мы отдаем TomCat'ом. Но с нативным коннектором. В принципе, мы вполне им довольны.

Вопрос из зала: У меня вопрос связан с предыдущим, который был по поводу построения ссылок. То есть перед вами не стоит задача защищать ссылки, они не авторизованы, скажем так?

Александр Христофоров: Ссылки на скачивание фотографий?

Вопрос из зала: Да.

Александр Христофоров: Нет, у нас можно заходить, качать.

Вопрос из зала: Имеется в виду, если пользователь закрыл фотографию и открыл ее только для своих друзей, то, получив ссылку на эту фотографию, не являясь другом пользователя, я смогу ее посмотреть?

Александр Христофоров: В общем, да. Честно говоря, приватность у нас — достаточно новый сервис. Я даже не знаю, в какую сторону он развивается. (Смех, аплодисменты в зале). Нет-нет. Я к тому, что я, правда, даже не знаю, как он работает. Я не пользовался приватностью на «Одноклассниках», поэтому я не знаю, можно ли там защитить фотку.

Вопрос из зала: Понятно. Спасибо.

Вопрос из зала: То есть, в принципе, у вас может возникнуть такая проблема (неразборчиво).

Александр Христофоров: На самом деле, это не особая проблема. Все равно мы ходим всегда через TomCat. Даже если надо защитить фотку — Ok, построим авторизованную ссылку, проверим ее на TomCat'ах. Точно так же мы делаем сейчас при аплоаде. При аплоаде мы проверяем, у нас ссылка авторизованная. Это очень дешево сделать, это очень легко.

Нестандартное использование репликации Mysql

Дмитрий Самиров, Александр Панков

Александр Панков: Мы расскажем чуть-чуть о нестандартном использовании MySQL-репликации для написания высоконагруженных демонов. Покажем, как это все должно работать правильно. Я думаю, что кто-то из вас уже сегодня сможет посмотреть наши открытые исходники, наши открытые примерчики и посмотреть, как можно свою систему спасти от большой нагрузки.

Я вам представлю Дмитрия Самирова. Это разработчик серверной части нашей рекламной системы Advection. Безумно талантливый. Вовсю использует MySQL-репликацию.

Нестандартное использование репликации Mysql



- Используем Mysql как проверенное временем решение;
- Ускоряем систему демонами на языках низкого уровня;
- Используем репликацию Mysql для синхронизации;

Вот такая петросян-картиночка небольшая, чтобы всем было хоть капельку смешно.

Я, в принципе, никогда не любил классические базы данных, потому что чаще всего в высоких нагрузках от них только проблемы и никакого толку. Тем не менее, web-интерфейс с использованием стандартного MySQL на PHP пишется очень дешево и очень быстро. Поэтому полностью отказаться от баз данных невозможно. Приходится поступать так: пишутся web-приложения, которые работают непосредственно с базой данных, а высоконагруженная часть постепенно как-то перекочевывает на какие-нибудь самописные демоночки и прочие штуки.

Простейшая схема работы с БД

Обработчик
запросов

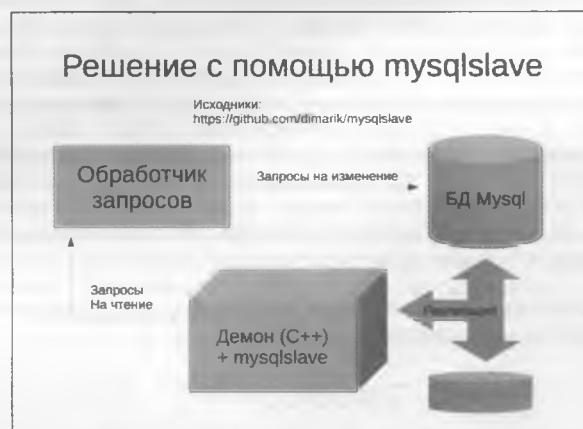


БД Mysql

На слайде — пример типичного сайта на Drupal, на Joomla, на Bitrix. Типичный сайт, работает с базой данных, и на него ходит 50 человек в день. Все хорошо работает.



Классическое решение на Memcached. Какой его основной недостаток? Во-первых, обработчик запросов идет всегда в Memcached, смотрит, есть ли там данные. Если нет, то обращается в базу, и все в принципе всех устраивает. Одна фигня: если данные изменились, в Memcached останется старая версия. Спрашивается: зачем она тогда там нужна. Получается, обработчик должен следить за всеми Memcached и постоянно обновлять Memcached. Сложно отследить, поэтому на многих сайтах показывается неактуальная информация. Видно сразу, что что-то не то закэшировали.



На слайде архитектура, которую используем мы. Она прекрасна тем, что, во-первых, быстро высоконагруженная часть обслуживается демоном на C++, в котором всегда гарантированно данные актуальные и правильные.

Как это все устроено. Есть база данных, в которой, мы считаем, что данные самые актуальные. Это master база либо slave первого уровня от master базы. Неважно. Мы с помощью репликаций, с помощью нашей библиотечки, исходники, которые вы можете скачать уже прямо сейчас и посмотреть живой пример, как это все работает.

Демон представляется базе данных, как будто бы он другой MySQL и просто-напросто следит за обновлениями базы. Что мы в результате получаем. То, что стартанув этот демон, вычитав

фактически всю базу и быстро ее раздавая, мы получаем всегда свежую и актуальную копию. Репликация доходит быстро.

Как это выглядит в демоне?

- Создаём класс, наследующийся от mysql::CLogParser;
- Реализуем 3 виртуальных метода — on_insert, on_update, on_delete;
- Инициализируем: set_connection_params(...);
- watch("db1", "table1"); // для каждой таблицы
- prepare();
- В отдельном потоке запускаемся.

```
static void* __replication_thread_proc(void* p)
{
    test_demon* _d = (test_demon*)(p);
    _d->replication_thread_proc();
    return 0;
}
```

Дима сейчас расскажет, как это все используется, а используется это очень просто.

Дмитрий Самиров: С одной стороны, может показаться, что здесь нужно что-то думать, чтобы это все применить. На самом деле, это не так. Используется все очень просто. Здесь написаны какие-то слова. Я не буду к ним притираться, потому что какая-то часть кода. Вообще на GitHub есть ReadMe, и там довольно подробно написано, как это все используется. Все довольно понятно.

Я расскажу просто в двух словах, как это используется. Пусть у вас есть какой-то демон, и вы хотите, чтобы он следил за репликацией. Вы просто берете класс этого демона и добавляете, чтобы он еще наследовался от нашего mysql::CLogParser. В этом базовом классе содержатся некоторые виртуальные функции (конкретно здесь написано). Эти функции вызываются, когда в базе происходят изменения, и происходит репликация.

Дальше еще нужно в наш класс внести еще несколько строчек кода, чтобы банально инициализировать, подконтактиться к MySQL. Как уже сказали, он коннектится просто как обычный пользователь. MySQL-сервер просто видит, что это еще один slave client. Мы должны перечислить таблицы, за которыми мы будем следить. На этом использование заканчивается. Оно заключается в реализации этих трех функций.

Александр Панков: Собственно, Дима все кратко и рассказал. Создаем отдельный тред, он сидит, слушает репликации и обновляет весь демон.

Архитектурные особенности

- Только row-based replication;
- Блокирующийся на чтении поток;
- Версия 5.1 и выше;
- Зависимость от mysqlclient;
- Другие сюрпризы MySQL;
- Никаких зависимостей кроме STL (нет и не будет boost и пр.);

Архитектурные особенности. Только row-based репликации мы не умеем парсить SQL-запросы, и не надо этого делать. Row-based репликации приходят готовые строчки. Особенности нашей реализации: она реализована стандартно на базе MySQL-клиента. Из-за этого она висит на блокирующем потоке, на чтении (MySQL save cli), и приходится создавать отдельный поток. Мультиплексом пока не работает.

Версия 5.1 очевидно выше, потому что row-based репликация. Есть и другие реализации подобной штуки. В «Бегуне» есть реализация, но она использует Boost. Мы написали без Boost и без прочих зависимостей. Мы только используем чуть-чуть STL. Самую малость. Там почти все на нем.

Особенности настройки и работы

- Права доступа (GRANT SUPER, REPLICATION CLIENT, REPLICATION SLAVE, SELECT ON ...);
- Row-based replication;
- Автоматическое переподключение при правильном сбросе соединения;
- Требуется перезапуск при исчезновении соединения (ситуация kernel panic сервера БД);

Особенности работы. Как и все с MySQL. Работает — значит, работает. Не работает — значит, граблей может быть миллион. Все с MySQL именно так. Права доступа нужно проследить, чтобы были правильные.

Select нужен ровно лишь для того, чтобы ваш демон мог заселектить в самом начале структуру таблички и названия полей. Больше ни для чего select не нужен. Если ваш сервер перестартится как-то, все хорошо, демон это понимает. Соединения он восстанавливает после переброски.

Единственная штука, за которой стоит следить, это пропажа сервера. Сервер взяли, сетевой провод вытащили — сервер пропал. Демон репликации будет думать, что обновления просто не приходят. Этую особенность того самого MySQL-клиента. Мы никак не можем пропинговать сервер. Это небольшой недостаток.

Нечасто сервера сейчас в Kernel Panic уходят, и нечасто сетевые провода выдергивают. Тем не менее, лечится очень просто: перезапустили демона — и все восстановится. У нас пока еще этого не происходило, теоретически возможность рассмотрели.

Попробовать

<https://github.com/dimarik/mysqlslave/>

Исходники библиотеки и пример
(читаем README + TRY)

Попробовать, Дима уже говорил, читаем ReadMe. Есть на русском, есть на английском. Есть файл Try. Дима все написал, можете ему вопросы задавать, можете мне. Но я, скорее всего, не отвечу.

Будущее

- Тестирование на различных версиях;
- Убрать зависимость от mysqlclient, собственная реализация протокола;
- ...мелкие улучшения...

Будущее. Тестиовать надо на разных версиях MySQL. С версиями там беда: пишет непонятно кто, пишет непонятно как, и вообще его будущее туманно и непонятно. Тестировали на 5.5 (я тестировал), на 5.1 он работает, на 6.0 — я вообще понятия не имею, что это такое. Но, в принципе, никаких сложностей не должно возникнуть.

Будущее за отказом от MySQL-клиентской библиотеки. Исходники MySQL ужасны, и нам пришлось в эту прекрасную библиотечку перетащить часть этих исходников, в частности различные H-файлики, и ничего не получилось пока с этим сделать. Но Дима обещал написать свой клиент MySQL, который можно будет использовать мультиплексом, и все будет хорошо. Но до этого нужно просто реализовать бинарный протокол MySQL.

Мелкие улучшения. Почему я написал «мелкие», потому что все готово, можно использовать сейчас. Это production. Можете сейчас использовать, допиливать напильником придется в любом случае, потому что под вашу задачу... Ваш демон. Сама библиотека работает. Если что — допиливается просто.

Спасибо за внимание!

Дмитрий Самиров (dimas@advaction.ru)
Александр Панков (pianist@advaction.ru)

Рекламная система Advaction

Спасибо за внимание. Больше не будем вас грузить. (Аплодисменты). Вопросы можете задавать. Мы с удовольствием ответим.

Вопросы и Ответы

Вопрос из зала: (Неразборчиво, без микрофона, 10:35).

Александр Панков: Суперпривилегия нужна. Ну, MySQL какой-то, не знаю... Не дашь супер-привилегии — и репликации не получает.

Вопрос из зала: (Неразборчиво, без микрофона, 10:44).

Александр Панков: Значит, не нужна. Но мы ему дали, нам не жалко.

Вопрос из зала: Почему демон не может узнать об отваленном сервере?

Александр Панков: Он может узнать. Если сервер рестартится, то все хорошо, он узнает. Сервер закрывает сокет, и все узнается. Если сервер исчезает (такая ситуация возможна — провод выдернули), демон никак об этом не узнает.

Вопрос из зала: MySQL_ping.

Александр Панков: Он не может MySQL_ping сделать по одной простой причине.

Вопрос из зала: Если сервер исчезает, почему его нельзя его проверить MySQL_ping либо обычным ping'ом?

Александр Панков: Потому что слушающая репликацию штука, написанная в MySQL-клиенте, висит в Read'e банально. Она просто ждет события от сокета. Должно произойти что-то. Сокет должен кто-то закрыть. Его некому просто закрыть. Такова репликация MySQL.

Вопрос из зала: Он же по Time out отвалится.

Александр Панков: Time out'ов там нет. Кстати говоря, в этом беда. С какой стати должен произойти Time out. Time out чего? Сокет на этой стороне живой. Он нормальный. Может быть, через полгода сервер проснется и перешлет данные. Увы, так устроен MySQL.

Вопрос из зала: То есть фоном не проверяется наличие сервера никак?

Александр Панков: Мы, наверное, будем это делать ровно после того, как напишем свою клиентскую MySQL библиотеку. Перестанем использовать тот треш, который официальный MySQL из себя представляет.

Вопрос из зала: Скажите, Keep-Alive не помогает в такой ситуации?

Александр Панков: Не помогает, естественно.

Вопрос из зала: Демон вообще физически, данные где-то хранит?

Александр Панков: Я еще раз вернусь к слайду, чтобы показать, как работает демон.



Самый тупой, типичный пример. При старте демон вычитал всю базу и хранит в памяти, просто получает апдейты. Если таблица большая, то, скорее всего, тоже можно это сделать. Но можно просто расшардить 10 демонов.

Вопрос из зала: То есть демон грузит при старте?

Александр Панков: Например, так. Можно при запросах. У нас есть демона, которые при запросах, если у них нет данных, они делают select. Если есть данные — они отдают из кэша. Этакий аналог Memcached. Такой демон возможен. У нас такая реализация есть. Делаете дополнительный select. Просто мы любим однопоточное. Однопоточное select не может сделять.

Вопрос из зала: Тогда еще последний вопрос. Раз Row-based пользуется, предположим, у нас произошел запрос на изменение. Безусловный апдейт. В этой табличке миллион строк. Вы получили миллион Row-based сообщений — и у вас сразу же лаг. В любом случае.

Александр Панков: Это не страшная штука, потому что наши демоны хорошо написаны. Они этот миллион так: фить — и все. Пролетит. У нас нет проблемы с обработкой миллиона запросов.

Вопрос из зала: Миллион запросов надо же где-то держать, их надо куда-то писать, в память.

Александр Панков: Я не знаю. Пока проблем не было. Мы обратим, обязательно, на это внимание. Попробуем сделать какой-нибудь апдейт. Но наши демоны нормально получают.

Вопрос из зала: А если DDL-запрос спускается, вы что будете делать?

Александр Панков: По поводу сетевой части. У вас есть обычный master и обычный slave. Какие они проблемы с сетью ограбут, такие же будут проблемы с сетью с демоном. Какие они проблемы не ограбут, таких же проблем не будет с демоном. Ничего принципиально нового нет. Демон ведет себя, как обычная база данных. Там Copy-Paste исходников.

Вопрос из зала: В чем профит тогда?

Александр Панков: Профит? В демоне всегда свежие данные.

Вопрос из зала: Миллион строк?

Александр Панков: Миллион. Ну, и что?

Вопрос из зала: Лаг же все равно будет такой же, как и на том обычном slave'е. Сиюсекундно он же этот миллион не запишет.

Александр Панков: Ну, с каким-то минимальным отставанием. А вы хотите сказать, что тут принципиально другое возможно решение?

Вопрос из зала: Мне, как раз, интересно, чем ваше решение отличается от стандартной репликации.

Александр Панков: Наше решение — это как раз нестандартное использование репликации. В репликации ничего не меняется. В нашем решении меняется следующее: стандартная репликация делает практически копию базы данных — мы же обновляем данные в демоне. Просто для этого написана специальная библиотека, достаточно удобная. Больше ничего.

Вопрос из зала: Хорошо. Спасибо.

Вопрос из зала: Спасибо за доклад. Я посмотрел на GitHub у вас через три функции реализовано On_insert, On-delete, On_update. Как вы границы транзакции видите вообще?

Александр Панков: Там они приходят как-то, но... (Смех). На самом деле все хорошо. Смотрите, какая фигня происходит. Границы транзакции мы не видим, и нам их не нужно видеть. Master, до тех пор пока коммит ему не скажут, не будет это все рассыпать. Репликация до нас не дойдет до тех пор, пока не будет сделан коммит. Если коммит сделан, то мы получим все update.

Вопрос из зала: Это все хорошо, но если вы не видите границ транзакции, вы не знаете точки, в которых данные консистентны.

Александр Панков: События Begin и Commit, повторюсь, они приходят. Необходимости в этом не было. Написать функции On_begin, On_commit — это хороший патч. Можете сделать его сами, закоммитить на GitHub, у Димы попросить доступ. Можете попросить — сами закоммитим. Сложностей с этим нет, потому что это простые события: On_begin, On_commit. Они приходят. Они даже по логам видны.

Вопрос из зала: Спасибо за предложение. Я просто работаю в компании «Бегун», у которой тоже выложена библиотека libslave с этой функциональностью. Спасибо.

Александр Панков: Но у вас она использует Boost — это ужасно, мне кажется.

(Смех в зале).

Вопрос из зала: А все-таки, если спускается DDL-запрос, как-то это обрабатывается?

Александр Панков: Я не знаю, что такое DDL, простите.

Вопрос из зала: Alter.

Александр Панков: А, с Alter Table, на самом деле, все хорошо. Alter Table тоже приходят, мы их игнорируем. (Смех в зале, аплодисменты). Обычно такое изменение логики требует перезапуска демонов. Но, действительно, подумайте сами. Ну, сделали Alter Table. Наверняка, там, в демоне, придется перефигачить половину всего, скорее всего.

Вопрос из зала: Просто добавили индекс.

Александр Панков: Добавили индекс? Он проигнорирует и улетит спокойно. То есть, а демон...

Вопрос из зала: Проигнорирует?

Александр Панков: Индекс? Зачем ему знать о существовании индекса?

Вопрос из зала: Как демон тогда вообще отдает данные? Он же их тоже, по идее, селектит из своей базы, или как?

Александр Панков: Я могу сказать, что для того демон и написан, чтобы он внутри уже сам решал, как ему отдавать данные, как ему селектить, какие индексы.

Вопрос из зала: То есть он на индексы не смотрит?

Александр Панков: Ему на это вообще пофигу.

Вопрос из зала: К нему запросу SQL'ем приходят?

Александр Панков: Нет. HTTP'шным, допустим. Я приведу типичный пример. Мы реализуем демон. Демон, допустим, профайлов пользователей на форуме, в «Одноклассниках», очередной социальной сети. Реализуем демон профилей, которому говорим: «Дай мне профиль такого-то человека». Если он профиль этого человека не имеет, он селектит из базы из одной, из другой из третьей-пятой-десятой, составляет внутреннюю структуру, в которой это все за-кешировано и отдает JSON'ом.

Приходят апдейты. Допустим, у него друзья добавились или фотки новые — не знаю, что угодно. В базу пришел апдейт. Он видит. Он просто корректирует эту внутреннюю структуру данных. Демон в данном случае — это не структура данных хранения, это структура данных использования. Оно в памяти, скорее всего, будет, он используется не как диск. Если он диск — зачем SQL, вообще?

Вопрос из зала: То есть он не сможет создать сложную структуру.

Александр Панков: Как не сможет?

Вопрос из зала: Мне нужно получить результат — куча Join'ов.

Александр Панков: Если вы пропишете все эти Join'ы честно в демоне... Опять же. Это все надо реализовать внутри демона. Это не прослойка между базой, заменяющая SQL-запросы. Это штука, которая позволяет типовой запрос, допустим, тысяча, две, три, пять в секунду просто кешировать и автоматически обновлять.

Вопрос из зала: Все. Тогда понятно. Спасибо.

Александр Панков: Есть еще у кого вопросы? Все, спасибо.

Секция «Системное администрирование»

Принципы балансировки

Алексей Бажин

Ведущий: Ведущего секции не нашлось, поэтому буду работать немножко я. Я докладываю через один доклад.

Представляю Алексея Бажина. Это главный системный администратор Mail.ru. Он немного расскажет про такую редко используемую вещь, как балансировка. Редко используемую — в смысле, в маленьких компаниях. Про то, что это вообще такое, как бывает, как это используем мы. Про остальные секции тогда потом.



В данном докладе я хочу рассказать о принципах балансировки для внешних сервисов с неспециализированными клиентами. Самый простой пример такого сервиса и клиента — это web-сервер и браузер. Мы рассмотрим сами принципы балансировки, их плюсы и минусы, немножко поговорим об их реализациях и применимости их к различным сервисам.

2010

Какие задачи может решать балансировщик

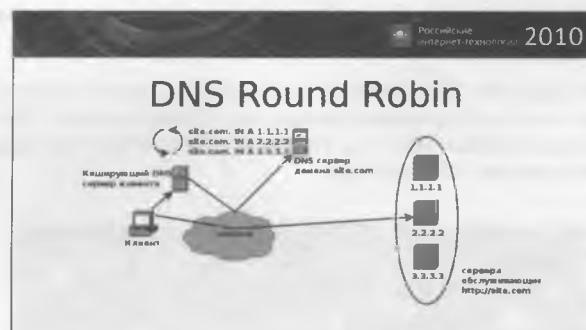
- Распределение нагрузки между серверами
- Повышение отказоустойчивости
- Защита от некоторых типов атак

Допустим, у нас есть некоторый web-проект, web-сервер которого пока что живет на одном сервере. Тут мы уже начинаем упираться в его производительность. Самое очевидное решение в данном случае — это апгрейд «железа». Но если мы пойдем по этому пути, то рискуем через некоторое время упереться в то, что апгрейдиться будет просто-напросто не на что.

Тут мы понимаем, что нам нужно каким-то образом кластеризовать наш проект. Сама кластеризация несколько выходит за рамки моего доклада. Поэтому будем считать, что мы это сделали.

Теперь у нас встает задача, каким образом мы будем распределять нагрузку между нашими серверами. Для этого мы и будем использовать балансировку. Кроме самого распределения нагрузки мы, возможно, хотим решить еще ряд некоторых других задач.

Например, повышение отказоустойчивости (то есть, чтобы при выходе из строя одного из серверов у нас проект продолжал работать) и защита от некоторых типов атак. Например, от открытия большого числа соединений, в которых ничего не говорится.



Самый простой метод балансировки — это DNS Round Robin. Суть его в том, что на DNS-сервере для записи нашего домена мы создаем несколько DNS-записей типа A. DNS-сервер выдает эти записи в чередующемся циклическом порядке.

Обычно на клиентах resolver устроен таким образом, чтобы кэширующий DNS-сервер клиента не влиял на нашу балансировку. Клиент еще и сам из полученных записей выбирает случайную. Соответственно, происходит соединение с нужным сервером.

Для реализации данного метода подойдет совершенно любой DNS-сервер. Допустим, BIND (он же Named).

The slide has a dark header bar with the text 'Российский интернет технологии 2010'. The main title 'DNS Round Robin' is centered in a large font. Below it, there are two sections: 'Плюсы:' (Pros) and 'Минусы:' (Cons), each listing several bullet points.

Плюсы:

- Не зависит от протокола высокого уровня
- Не зависит от нагрузки
- Не требует дополнительных ресурсов
- Не требует связи между серверами
- Низкая стоимость решения

Минусы:

- Возможно неравномерное распределение нагрузки (например при наличии клиентов на Windows Vista)
- Сложно отключать не отвечающие сервера
- На практике максимальное число IP-адресов ограничивается размером UDP пакета в DNS

Из плюсов этого метода:

- Он абсолютно не зависит от протокола высокого уровня. То есть любой протокол, где обращение к серверу идет по имени, может использоваться для него.
- Также он не зависит и от нагрузки на сервер. Благодаря тому, что есть кэширующие DNS-сервера, нам все равно, сколько у нас будет клиентов — хоть единицы, хоть миллионы.
- Также он не требует связи между серверами. Поэтому может использоваться как для локальной балансировки (это балансировка серверов внутри одного DC, скажем), так и для глобальной балансировки, когда у нас есть несколько DC-центров, где сервера между собой никак не связаны.
- Самый главный его плюс — это низкая стоимость решения. Если у нас уже есть проект, домен, DNS-сервер, все, что нам нужно, чтобы перейти к этому методу балансировки, это всего лишь добавить еще записей в DNS.

Из его минусов можно отметить:

- Сложно отключать сервера, которые не отвечают или вышли из строя. В DNS существует то же самое кэширование. Мы запись убрали, а она доедет до правильных клиентов через время, которое задается TTL (Time To Live) в DNS-зоне.
- Бывают еще не очень правильные DNS-серверы у некоторых провайдеров, которые принудительно кэшируют записи на гораздо более долгое время. Мы встречали даже такое: убираешь запись из DNS, а клиенты туда еще год продолжают ходить. Такое нам тоже встречалось.
- Следующая вещь, которую мы заметили. В данном методе очень сложно распределять нагрузку между серверами в нужной пропорции. Единственный способ это сделать — навесить на каждый сервер по несколько IP-адресов так, чтобы их количество было пропорционально нужной части нагрузки, которая должна на них идти. Это минус, так как IP-адресов у нас обычно не очень много.
- Следующий минус — возможно неравномерное распределение нагрузки благодаря стараниям компании Microsoft. Дело в том, что Windows 2008 и Windows Vista без Service Pack'ов resolver по умолчанию настроен таким образом, что он из полученных от сервера записей выбирает не случайную запись, а ту, которая кажется ему ближе по номерам сети к его собственному IP-адресу.

Если у нас IP клиента 1.1.1.1, а сервера есть 2.2.2.2 и 3.3.3.3, то будет выбран 2.2.2.2, так как он ближе. Из-за этого получается, что... В Рунете у нас сеточки собраны только в определенных местах обычно. Проект для аудитории Рунета. Получается, что если мы используем DNS Round Robin, то все такие клиенты идут на один и тот же сервер. Это может давать довольно неслабую неравномерность при распределении нагрузки между серверами.

Это, конечно, было наиболее актуально около года назад. Сейчас уже от Microsoft вышли исправления для этого. Но до сих пор еще заметно.

Интересно, как это нами обнаружилось. В одном из проектов, где использовалась Round Robin балансировка DNS, у нас случилась проблема на одном из балансируемых серверов. Как оказалось, все компьютеры с Vista из нашего офиса ходили как раз на этот сервер. После этого мы начали выяснять и выяснили такую вещь.

Следующий минус. На практике максимальное число IP-адресов, которые мы можем балансировать таким методом, оказывается ограниченным. В протоколе DNS в случае, если ответ превышает 512 байт, выдается по IDP отбивка, что нужен перезапрос по TCP.

Как оказалось, не на всех DNS-серверах открыты обращения по TCP от клиентов. Например, года полтора назад мы столкнулись с такой проблемой, что оно было закрыто у «Стрима». Со «Стримом» мы проблему довольно оперативно решили, но он оказался не единственным. Поэтому при больше, чем 20 с небольшим записях DNS Round Robin мы тоже не смогли применять.

Учитывая, что DNS является одним из самых удобных методов глобальной балансировки, то появилось желание каким-то образом обойти эти проблемы.

2010

А можно по-другому?

Можно:

- Используем короткий TTL
- Выдаем по одной записи на каждый DNS запрос

Плюсы:

- Отсутствие неравномерности при небольшом числе серверов

Минусы:

- Малый TTL записей (больше нагрузка на DNS)
- Принудительное кеширование на DNS серверах

Самое очевидное решение, которое пришло в голову: используя короткий TTL, выдавать на каждый запрос только по одной записи, чередуя их необходимым образом. Буквально за вечер был написан модуль для DNS-сервера PowerDNS, который и реализовал этот метод.

Плюсы этого метода:

- Мы решили проблему с неравномерным распределением нагрузки.
- Мы можем использовать Round Robin с весами в этом методе, выдавая нужный IP-адрес большее число раз.
- Можем распределять нагрузку с нужными нам пропорциями.

Минус — это малый TTL записи. Соответственно, большая нагрузка на DNS-сервера. Мы использовали TTL 1 минута. Еще было опасение, что...

Существуют DNS-сервера, которые обслуживают большое число клиентов. В определенный момент времени (в течение минуты Time To Live) они отдают один и тот же сервер всем клиентам. Из-за этого может быть неравномерность. Но как показала практика, даже при 30 балансируемых серверах такой неравномерности не наблюдается.



Следующий метод балансировки — это балансировка на втором уровне стек протоколов. У нее есть два варианта: с использованием отдельного выделенного балансировщика и без.

В обоих случаях мы берем некоторый IP-адрес нашего сервиса. Его мы навешиваем на все наши сервера (один и тот же) либо на ОВЕС, либо на другой специализированный интерфейс. Делается это для того, чтобы эти сервера могли принимать соединения на этот IP-адрес и отвечать с него, но не отвечали бы на ARP-запросы, относящиеся к этому адресу.

Как он работает? На балансировщик, имеющий этот IP-адрес и отвечающий на ARP, приходит, допустим, первый пакетик соединения. Мы смотрим, что он первый. Нужным алгоритмом отправляем его на интересующий нас сервер, подменяя MAC-адрес на destination. Записываем его в некоторую табличку соединений.

Если у нас это не первый пакетик, то мы просто смотрим по табличке соединений, каким сервером обрабатывается это соединение, и его туда отправляем.

Учитывая, что мы не модифицируем заголовки третьего уровня и выше, то ответы от серверов мы можем пускать прямо мимо балансировщика напрямую через Интернет клиенту (до нашего шлюза).

Из программных реализаций данного метода самое распространенное — это Linux Virtual Server. В URL-терминологии данный метод балансировки называется Direct Routing.



Существует еще один метод балансировки на втором уровне — без выделенного балансировщика. В данном случае наш IP-адрес сервиса мы прописываем как статическую ARP-запись на нашем шлюзе с некоторым мультикастным MAC-адресом. Настраиваем наши коммутаторы таким образом, чтобы фреймы, приходящие на этот MAC-адрес, доставлялись всем нашим серверам.

На серверах мы вычисляем некоторый HASH, скажем, от IP-адреса клиента. По его значению сервер понимает, должен ли он отвечать на эти запросы. Если HASH=0, то должен отвечать первый сервер. Он и отвечает. Остальные знают, что они не должны отвечать.

Балансировка на 2-ом уровне

Плюсы:

- Не зависит от протокола высокого уровня
- Есть методы без выделенного балансировщика
- Есть возможность пускать ответы мимо балансировщика
- Относительное малое потребление ресурсов

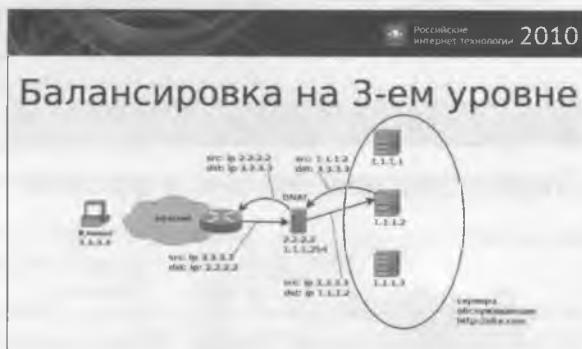
Минусы:

- Сервера должны находиться в одном сегменте сети
- Необходима специфическая настройка серверов и сетевого оборудования

Плюсы этих методов:

- Они также не зависят от протокола высокого уровня. Можем балансируировать хоть HTTP, хоть FTP, хоть SMTP.
- Есть метод балансировки без выделенного балансировщика. Если у нас небольшое число серверов, то для нас это может быть актуально.
- Есть возможность пускать ответы мимо балансировщика. Учитывая, что, допустим, в протоколе HTTP размер ответа обычно на порядок больше, чем размер запроса, то мы довольно сильно экономим на ресурсах.
- Из этого истекает относительно малое потребление ресурсов.

Очевидный минус этого метода — все сервера должны находиться в одном и том же сегменте сети. Необходима специфическая настройка серверов и сетевого оборудования. Поэтому этот метод может оказываться не всегда удобным и не всегда применимым.



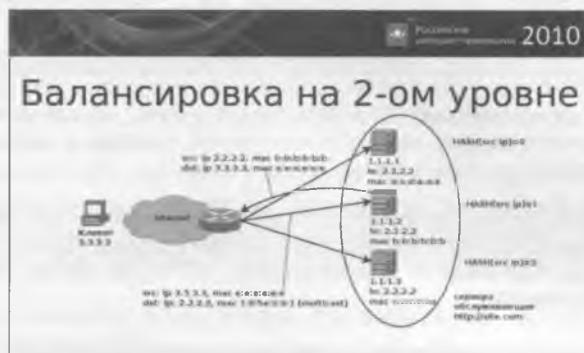
Очень похожий на него метод — метод балансировки на третьем уровне стек протоколов, то есть на уровне IP. В нем мы назначаем балансировщику тот же IP-адрес сервиса. Когда идет на него обращение, мы делаем так называемый Destination NAT, то есть подменяем в пакете IP-адрес назначения с IP-адреса сервера на IP-адрес сервера, который будет его обрабатывать (выбранный по нужному алгоритму).

Отличие его от предыдущего метода состоит в том, что мы уже модифицируем заголовки третьего уровня. Поэтому для ответов нам их тоже нужно обратно модифицировать. IP-адрес

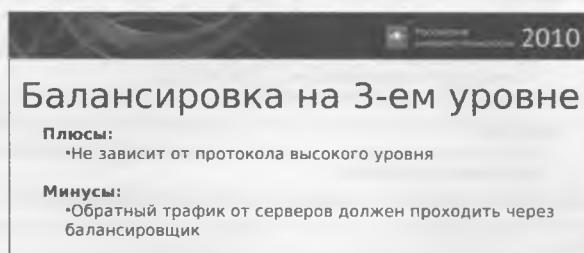
отправителя нужно поменять с IP-адреса сервера, который обрабатывает запрос, на IP-адрес сервиса, который висит на балансировщике.

Реализаций этого метода тоже достаточно много. Самые простые — тот же Linux Virtual Server, можно сделать над Iptables'ом. Существует еще множество аппаратных реализаций.

У нас в Mail.ru это наиболее часто используемый метод балансировки. Он не требует никакой дополнительной настройки сетевого оборудования или серверов. Он для них абсолютно прозрачен. Сервера видят, как будто к ним обращается сам клиент.



Забыл упомянуть, что в качестве реализации этой схемы можно использовать кластер IP в Linux'овом Firewall'e Iptables.



Плюсы балансировки на третьем уровне:

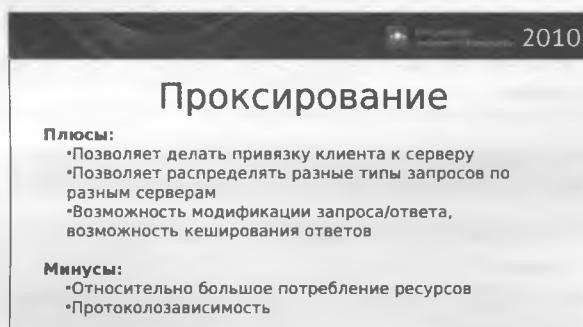
- Она не зависит от протокола высокого уровня.
- Она абсолютно прозрачна для серверов.

Минусы — обратный трафик серверов должен проходить через балансировщик. Соответственно, нагрузка на него будет несколько выше, чем при использовании балансировки на втором уровне.



Следующий метод — проксирование. Есть некоторый прокси, на котором висит IP-адрес нашего сервиса, он получает запрос, может с ним что-то сделать и перенаправляет его уже от себя нужному серверу.

Этот метод для сервера уже не прозрачен, потому что он видит, что к нему обращается наш прокси. Для этого внутри протокола высокого уровня мы должны каким-то образом передать, какой же клиент к нам обращается. Для протокола HTTP мы можем добавлять заголовок X-Real-IP с IP-адресом клиента.



Плюсы проксирования:

- Так как она работает на уровне протокола, то может анализировать и изменять наши запросы и ответы. Это позволяет делать привязку клиента к серверу. Допустим, по значению определенной cookie. Нужно это нам, если мы используем локальное хранение сессии данного клиента на сервере.
- Благодаря этому она позволяет распределять разные типы запросов по разным серверам. Мы можем выделить отдельные сервера, которые будут отдавать статически генерируемую главную страницу, и делать это с гораздо меньшим временем отклика, чем сервера, которые обрабатывают тяжелые запросы.
- Есть возможность модификации запроса-ответа. Там тоже добавление заголовка, например, перекодировка. Есть возможность кэширования ответов на нашем прокси.

Минусы:

- Из всех рассмотренных нами методов у проксирования самое большое потребление ресурсов, так как он идет по протоколам выше, чем все остальные.
- Для каждого протокола у нас должен быть свой тип прокси. Не для всех протоколов его можно реализовать таким образом, чтобы он решал нужные нам задачи.

Еще один метод балансировки — балансировка редиректом. Он применим для довольно малого числа протоколов. К счастью, для HTTP он применим.

Есть у нас некоторый балансировщик, который при обращении к нашему сервису (например, <http://site.com>) дает клиенту редирект, что нужно обратиться к конкретному серверу (например, <http://server2.site.com>). В случае HTTP это будет HTTP redirect 302, по-моему, moved temporary.

Плюсы этого метода:

- Если запросы достаточно тяжелые, то его иногда имеет смысл использовать и для глобальной балансировки. У нас есть балансировщик, который редиректами отправляет запросы на обработку в разные Data-центры.
- Также он позволяет распределять разные типы запросов по разным серверам. Так как запрос к нему приходит, то он вполне может его анализировать.

Минусы:

- Он, как я уже сказал, применим к очень малому числу протоколов высокого уровня.
- Для клиента на каждый запрос мы, получается, делаем два запроса. Один — к нашему редиректору, один — к серверу, который обрабатывает соединение. Это увеличивает время, через которое клиент получит окончательный ответ на свой запрос.

Из реализаций этого метода и предыдущего (проксирования) для HTTP можно применять всем известный web-сервер nginx.

Хочется сказать еще о том, какие есть алгоритмы распределения нагрузки между серверами.

Самый простой метод, когда мы высчитываем хеш от некоторых параметров нашего соединения. Допустим, от IP-адреса клиента, его порта. По значению этого хеша отправляем пакетики на нужный сервер. Минус этого алгоритма в том, что он должен быть довольно сложный, чтобы мы могли в случае исключения или включения серверов распределять нагрузку по другим.

Следующий метод — Round Robin с весами (простой Round Robin нам мало интересен). Это некоторое чередование наших серверов, когда частота их появления пропорциональна весу, который задан для данного сервера.

Самый интересный, наверное, метод балансировки — когда к нам приходит соединение, мы его отправляем на сервер, который в данный момент обрабатывает наименьшее число соединений. Соответственно, есть вероятность, что он быстрее всего обработает наше.



Из реальных схем балансировки хочется отметить, что иногда мы хотим увеличить отказоустойчивость, зарезервировав сам балансировщик.

В таком случае мы можем поставить два балансировщика, сделать некоторые резервирования между ними отдельные. Например, протоколами VRRP или CARP. Из линуксовых реализаций: UCARP, Keeper AFD, HardBit (?). Нагрузку между ними мы можем балансируировать DNS'ом, а сами балансировщики уже будут распределять нагрузку между серверами.

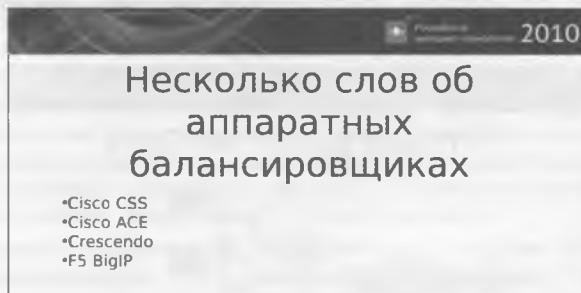
Это схема, которая используется на абсолютном большинстве наших проектов. Она применима для малого числа серверов. Допустим, есть у нас 2 сервера. Мы можем в качестве балансировщиков использовать прокси, которые будут висеть прямо на тех же серверах и резервировать их CARP или VRRP. Также эта схема применима и для большого числа серверов, когда стоят аппаратные балансировщики, а за ними — большие числа серверов.



Следующая интересная схема. Мы можем выделить отдачу статики на отдельные сервера, чтобы не пускать ее через наш балансировщик, который достаточно умный. Обработка каждого соединения для него достаточно ресурсоемка. Статика обычно отдается достаточно легко, малым числом серверов.

С этими серверами обычно ничего не случается, кроме того, что они могут просто падать. Нам их достаточно также зарезервировать каким-нибудь образом. Нам необязательно следить за их временем отклика. Так как это статика, то, скорее всего, он будет хороший.

Можем сэкономить большую часть ресурсов балансировщика на том, что много запросов к статике пойдут мимо него. Через него пойдут как раз тяжелые запросы, где он будет отслеживать число соединений с каждым сервером и все остальное.



Хотелось сказать несколько слов об аппаратных балансировщиках. На самом деле, я хотел сказать про них несколько больше. Но если рассказывать про них, то про каждый будет отдельный доклад. За мной будут выступать ребята из Crescendo, которые расскажут про свое «железо».

Из тех «железок», которые применялись или тестировались нами — это довольно старые CISCO CSS11503. Это некоторый аппаратный балансировщик с гигабитными портами, который позво-

ляет делать балансировку на третьем уровне и глобальную балансировку DNS'ом (в простом случае). У нас эти балансировщики до сих пор используются на не очень больших проектах.

Из их плюсов. У них очень легко прогнозировать нагрузку, которую они могут потянуть по тому, во что он упрется раньше: в пропускную способность сети либо по загрузке ЦПУ. Мы примерно знаем, сколько он может потянуть.

Более современное решение от CISCO — это модуль ACE для Catalyst 6500. Он умеет практически все вышеперечисленные методы балансировки, имеет теоретическую пропускную способность 16 гигабит.

Но с ним появляется некоторый ряд проблем, которые мы с аппаратом CISCO решаем не всегда очень успешно. Мы столкнулись с проблемой, что с ним довольно сложно спрогнозировать, какую нагрузку еще он сможет потянуть. **Вопрос:**

Из Crescendo мы тестировали «железку» CN-5510. Ее основное отличие от остальных в том, что она обрабатывает все полностью аппаратно. За счет этого имеет среди всех испробованных нами «железок» самое наименьшее время отклика. «Железка» эта тоже имела только гигабитные порты, к сожалению.

Сейчас мы тестируем решение BigIP от F5. Оно очень гибкое. Имеет свой встроенный скриптовый язык, который мы можем балансить, как нам захочется. Про него довольно подробно рассказывали на прошлом РутКонфе.

Мы потестировали некоторые его фичи. Но под большой нагрузкой его еще не тестировали, к сожалению. К следующему разу, может, сможем что-то рассказать.

На этом все. Я готов выслушать вопросы.

Вопросы и Ответы

Ведущий: Спасибо большое! Поблагодарим Алексея.

Вопрос: Здравствуйте. У вас было написано, при отделении статики вы используете несколько серверов, резервируя их методом CARP.

Алексей Бажин: Да. И балансируя DNS'ом.

Вопрос: Каким образом у вас контент... Дублируется, получается? Или как?

Алексей Бажин: Фактически да. Всю скачку мы просто складываем на отдельные сервера и отдаем поциальному доменному имени. Причем довольно полезно использовать имя из другого домена второго уровня. Это чтобы в запросы, которые идут туда, у нас не включались те же cookie, что идут на основной сервер. Чтобы размер запросов был меньше.

Вопрос: Я имею ввиду, вы контент... Или у вас лежит каждый контент на одном сервере? Упадет сервер — и Бог с ним. Или же именно дублирование?

Алексей Бажин: Да, он дублируется. Разумеется.

Вопрос: Storage'и не пробовали использовать? Дороже получается?

Алексей Бажин: Естественно. Если сравнить, что мы поставили два супермикровских сервера, которые могут обслуживать, скажем, третью часть Mail.ru. Их стоимость по сравнению со стоимостью storage'a несравнимая.

Вопрос: На Западе популярно использовать балансировщики для нагрузочного тестирования вживую. Вы не пробовали так?

Алексей Бажин: Я не совсем понял вопроса.

Вопрос: Когда нужно нагружочно протестировать какую-то часть проекта, вы просто балансирующим назначаете туда чуть больше нагрузку и смотрите, когда там начинаются какие-то...

Алексей Бажин: Да, мы используем иногда такой метод. Разумеется. Когда, допустим, приходит новое «железо», которое более производительно чем старое, мы таким образом можем посмотреть, сколько же оно вытянет.

Вопрос: Допустим, мы Round Robin'ом раскидывает на два IP: один-второй, один-второй, один-второй... Один сервер упал. Клиент попадает на сервер, заходит на сайт — получает ответ. Нажимает F5 — попадает на неправильный сервер, не получает ответ. F5 — на правильный, F5 — неправильный. Как с этим бороться?

Алексей Бажин: Как я уже рассказывал, можно использовать некоторые методы резервирования. Такие, как по протоколам CARP или VRRP. Для Linux'a — UCARP, Keeper AFD, HardBit (?).

Вопрос: По NAT-балансировке вы говорили про использование Iptables. Вы пробовали этот вариант?

Алексей Бажин: Да, пробовали. Но на практике мы его не используем, так как там очень сложно отключать упавшие сервера. Нужно что-то отдельно скриптировать. Только поэтому.

Вопрос: Там проблема, например, с контракт-таблицей, с размером соответствий между клиентскими...

Алексей Бажин: Размер контракт-таблицы задается sisKTL'ами (?). Это раз. Во-вторых, NAT-программы для очень больших нагрузок использовать не рекомендуется.

Вопрос: Если аппаратную, то какую порекомендуете? Какую-нибудь знает?

Алексей Бажин: Смотря какие нагрузки. Если до гигабита, то нас устраивает CISCO CSS. Если нужны большие нагрузки, то мы пока что еще сами в поисках. В принципе, тот же CISCO IS работает. Но есть нюансы.

Вопрос: Второй вопрос по поводу DNS-балансировки. У вас есть какие-то данные по поводу количества тех провайдеров, которые принудительно кэшируют записи? Это 1%, 2%, 5%?

Алексей Бажин: Скажем так. По истечении TTL'я 90% нагрузки уже идет куда нужно. Спустя сутки это уже, наверное, 98%. Оставшиеся 2% могут приходить довольно долгое время, как я уже говорил. Какая-то нагрузка видна даже спустя год бывает.

Вопрос: Какой метод посоветуете для глобальной балансировки с условием необходимости наличия привязки к сессии?

Алексей Бажин: Для глобальной балансировки очень сложно, точнее, не очень правильно делать привязку к сессии. Так как в глобальной балансировке можно использовать либо HTTP redirect. Тогда мы видим клиента, который к нам обращается, и можем его перекидывать куда нужно. Либо мы можем считать, что у нас каждый клиент приходит только с одного DNS-сервера. Соответственно, привязываться к этому. Но я бы не рекомендовал так делать.

Ведущий: У нас еще минута. На 1-2 вопроса можно ответить.

Вопрос: Вам не кажется, что с точки зрения большой компании ручками реализовывать через CARP и прочие вещи достаточно накладно? Пуще использовать какие-то решения от вендеров типа F5, CISCO. С точки зрения менеджмента, как вы считаете?

Я что имею в виду. Есть одна большая «железка». На ней, грубо говоря, один человек все раскидывает. Я правильно понимаю, что в случае, если мы пользуемся какими-то другими решениями, о которых вы рассказали в первой части, это будет достаточно большое количество узлов, зависимостей, человеческих ресурсов и рисков. Здесь можете пояснить?

Алексей Бажин: На самом деле, это так. Но учтем, что для нас в одну «железку» уже все не влезает. То есть у нас этих «железок» тоже получается немало. Это раз.

Во-вторых, для случая отдачи статики у нас съестся одна очень большая производительная «железка» только на это, когда мы можем вместо нее использовать всего 6 серверов, которые отдают статику.

Вопрос: Не верю. Ну, ладно.

Ведущий: Время вышло. Вопросов на момент предыдущего не было. Давайте перейдем к следующему докладу.

Управление памятью в гипервизоре. Все о виртуализации памяти в Parallels

Анна Воробьева

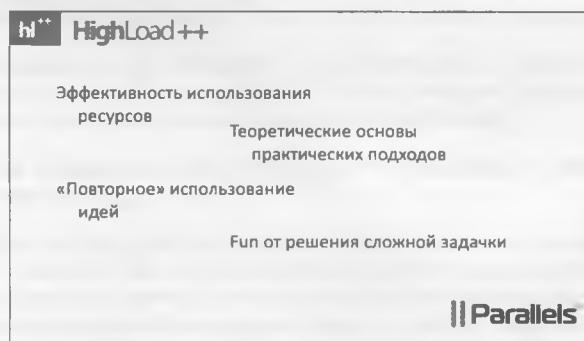


Анна Воробьева: Меня зовут Воробьева Анна, я занимаюсь разработкой, монитором виртуальных машин в компании «Parallels». Наша компания занимается десктопной и серверной виртуализацией. Мы делаем счастливыми Mac'овских пользователей, которые страдают от того, что не могут запускать Windows-приложения на своем любимом и замечательном Mac'e. Это один из наших флагманских продуктов. Для этого мы поднимаем целую виртуальную машину, обеспечиваем им интеграцию...

Но речь не об этом. Еще мы занимаемся развертыванием облачных сервисов для сервис-провайдеров. Но я хочу говорить о виртуализации. Точнее, я буду говорить о такой очень специфической теме виртуализации, как управление памятью.

Тема для меня очень интересная. Я могу говорить об этом очень много, но постараюсь себя фильтровать, лимитировать свой доклад, чтобы вам было интересно, и мы уложились в регламент.

Начну издалека. Начну с того, зачем нужна виртуализация.



Как-то так исторически сложилось, что когда-то можно было одно приложение пускать на сервере. У нас есть серверное приложение, мы ставим под него машину, и все отлично работает.

Потом сказали: «Давайте оптимизироваться, у вас under initialization». То есть одно приложение не полностью выедает ресурсы. Стали вводить виртуализацию. Мы стали увеличивать

количество виртуальных ресурсов на единицу физических ресурсов, увеличивая плотность. Мы увеличиваем плотность, создается состояния overcommit'a. Все хорошо, все счастливы, мы эффективно используем ресурсы.

По поводу overcommit'a по памяти существует большое количество мифов. Эти мифы говорят, что overcommit нестабилен; эти мифы говорят о том, что он недостаточно быстр, шустр, и что многие провайдеры не хотят использовать overcommit по памяти.

На мой взгляд, это вызвано прежде всего тем, что есть маркетинг. Маркетинг крупных компаний типа «Microsoft» говорит: «Все, что у нас не работает — неверно». У «Microsoft» overcommit по памяти не работает, поэтому «Microsoft» говорит, что он неверен. Это моя личная точка зрения.

Второй момент. Не всегда корректно конфигурируют. Кто-то что-то подумал, недодумал, и у нас получилась некорректная конфигурация, которая не дает максимальную производительность.

«Знания — сила». В этом докладе я хочу вам эту силу дать.

Я считаю, что идеи имеют тенденцию мигрировать из одной области в другую. Какая здесь у нас задача? Мы виртуализуем какой-то ресурс, мы распределяем ресурс в условиях дефицита между n-клиентами. Каждый из клиентов потребляет какое-то (разное) количество ресурса, которое нам заранее не известно. Это общая задача.

Я надеюсь, что те идеи, которые я здесь озвучу, могут быть успешно применены в вашей области. Кроме того, это достаточно интересно. Я попытаюсь последовательно ставить задачу, последовательно проводить вас от одного решения к другому, чтобы вы поняли, чтобы вам было интересно и чтобы вы получали максимальное удовольствие.

The screenshot shows a section of the HighLoad++ documentation. At the top, there's a header with the text 'hl++ HighLoad++'. Below it is a table of contents titled 'Содержание' (Content). The table of contents includes the following items:

- ✓ Постановка задачи
- ✓ Решения
 - ✗ Квоты, выбор backing store, алгоритма вытеснения
 - ✓ Balloon
 - ✓ Page sharing, compression
- ✓ Сравнение по продуктам

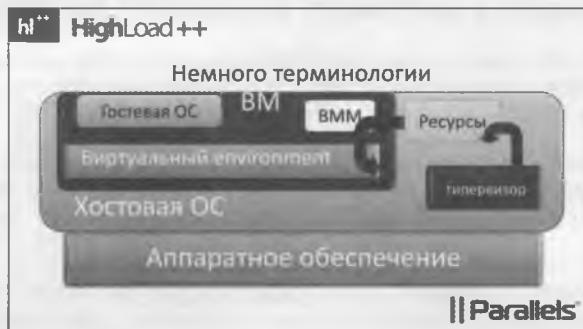
In the bottom right corner of the screenshot, there is a logo for Parallels.

Начну я, как бывший математик, с постановки задачи. Сначала я скажу о том, в чем условие, обозначу граничные условия, потом пойду по всяким технологиям. Я их буду вводить постепенно, говорить о том, в чем минусы, плюсы, почему одной технологии не достаточно и мы вынуждены применять другую. Закончу я все сравнением по продуктам.

Это очень субъективная штука. Меня попросили организаторы «HighLoad». Мы собирались перед «HighLoad» и рецензировали друг друга. Мне сказали: «Давай ты скажешь о том, как использовать это в продуктах?» Я скажу, но это мое личное мнение, поэтому тухлые помидоры, если они у вас остались, куда-нибудь спрячьте.

У нас есть аппаратное обеспечение. Это то, что у нас есть всегда, с самого начала. На нем мы разворачиваем host'овую операционную систему, или гипервизор. Гипервизор может быть отдельно, может быть компонентом host'овой операционной системы. Для нас это некий потребитель ресурсов, который распределяет ресурсы между всеми виртуальными машинами.

Что такое «виртуальная машина»? У нас есть монитор виртуальной машины — ВММ.



Он создает виртуальное окружение. На этом виртуальном окружении мы ставим гостевую операционную систему, которая знать не знает, что ее environment ненастоящий, и счастливо работает. Все это вместе — гостевая ОС, ВММ, виртуальный environment — в нашем понимании будет единой виртуальной машиной. Мы говорим об этом, как о потребителе ресурсов.



Задача распределения памяти выглядит примерно так. У нас есть n ВМ. Каждая из них потребляет какое-то количество ресурсов. Пока ресурсов достаточно, все хорошо. Но как только у нас создается дефицит ресурса, то нам нужно как-то разделить этот ресурс между всеми виртуальными машинами.

Мы можем пропорционально уменьшить ресурс для всех виртуальных машин. Выглядеть это будет ужасно. Все будут недовольны. Представьте: виртуальная машина 2 — это ваш любимый Exchange server (Apache), а виртуальная машина 3 — это какая-то тестовая нода, которая собирает статистику.

Мы увеличиваем у одного, отбираем у другого. Уже не все счастливы, но уровень счастья по больнице становится больше. Наша задача — достигнуть максимальной эффективности для всей ноды, для всех виртуальных машин, в соответствии с заданными ресурсами.

hi HighLoad++**

Разграничим термины

Overcommitment <ul style="list-style-type: none"> ✓ Σ{сконфигурированной памяти} + накладные расходы \geq разрешенный лимит ноды ✓ ВМ подлежат всем действиям, описанным в докладе 	Overload <ul style="list-style-type: none"> ✓ Σ{используемой памяти} + накладные расходы \geq разрешенный лимит ноды ✓ ВМ подлежат миграции
--	---

|| Parallels

Разграничим термины. Я говорила про дефицит ресурса, но иногда дефицит настолько сильный, что сделать уже ничего нельзя. Мы называем эту ситуацию «overload», когда весь ресурс вырабатывается целиком. Машины уже не могут работать, они должны быть мигрированы на другую физическую ноду. Здесь я даже какие-то формулы привела.

hi HighLoad++**



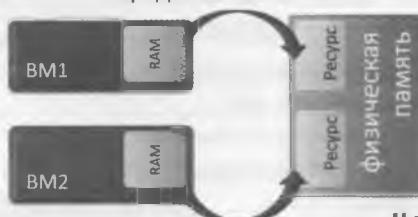
Overload

|| Parallels

Если говорить образно, overload выглядит примерно так. Это одна из причин непопулярности overcommit'a. Люди конфигурируют сервера так, что происходит overload, а после этого возмущаются, что скорость падает.

hi HighLoad++**

Распределение памяти: шаг 1



|| Parallels

Ваше внимание к этому слайду. Он сейчас такой маленький и пустой, но постепенно я буду его усложнять, накладывать на него дополнительные технологии.

У нас есть виртуальные машины, им назначена какая-то память. Этой назначенной памяти соответствует ресурс. Ресурс не равен памяти. Как мы уже говорили, ресурса не хватает.

Тем из вас, кто заканчивал какие-нибудь IT-подразделения каких-нибудь институтов, эта картинка должна напомнить курс теории операционных систем. У вас там есть процессы, каждый из которых потребляет виртуальную память — 2, 3, 4, 64 гигабайта, — а операционная система распределяет этот ресурс. Здесь то же самое.

Первый момент: если вы хотите использовать overcommit, внедряете новую технологию, то вы используете технологию из смежной отрасли.

hi** HighLoad++

Алгоритмы вытеснения

- ✓ LRU (last recently used)
- ✓ FIFO (first in first out)
- ✓ Aging (+NFU)
- ✓ NRU (not recently used ~ A-/D- bits)
 - а ведь еще можно дать всем второй шанс
- ✓ Clock
- ✓ Random

|| Parallels

Поэтому первое, что все делают, — внедряют алгоритмы вытеснения. Алгоритмов вытеснения очень много. Это last recently used (LRU), first in first out (FIFO), not recently used, not frequently used, aging. Можно еще дать второй шанс. Когда вы вытесняете страницу, вы ее не сразу откладываете в backing storage, а даете ей второй шанс. Пусть она еще побудет, а мы ее каким-то образом подкачаем.

Самое удивительное здесь в том, что есть алгоритм вытеснения произвольной страницы. Вы берете и вытесняете без всяких алгоритмов совершенно произвольную страницу. Самое забавное, что это коммерческое решение. «VMware» используют random. Они просто вытесняют произвольную страницу. Хорошие результаты.

Про алгоритмы вытеснения могу минут пятнадцать вам рассказывать, повторить школьный курс. Но я не буду этого делать. Почему? Потому что основная вещь, которую вы должны знать про алгоритмы вытеснения, состоит в том, что алгоритмы вытеснения не работают.

hi** HighLoad++

Алгоритмы вытеснения

не работают

- ✓ Гостевая ОС вытесняет страницы по своим алгоритмам (semantic gap)
- ✓ Отсутствие локальности обращений
- ✓ ОС не может поместить в процесс своего агента, а мы можем

|| Parallels

Почему они не работают?

Первое. Так называемое semantic gap. Что это такое? Вы такая умная виртуальная машина. Вы очень умная, вы все знаете про операционную систему, про ее железо. Вы начинаете каким-то образом вытеснять страницы. Вы обнаружили, что Windows не использует страницу, и ее себе куда-нибудь за ухо положили.

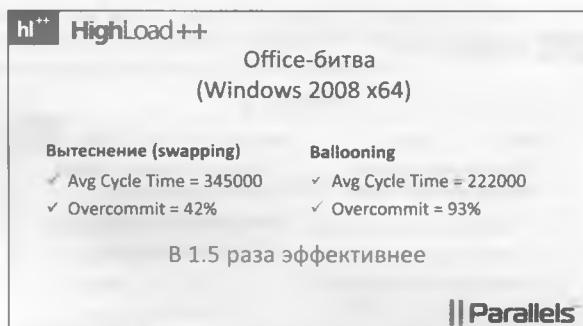
Windows не совсем глупая. Она тоже начинает вытеснять эту страницу. Как она ее вытесняет? Она говорит: «У меня страница не используется, обнулю-ка я ее». Обнуляет, подсасывает в себя страницу. Вам — накладные расходы и никакого удовольствия.

Отсутствие локальности обращений. Все алгоритмы, представленные на предыдущем слайде, исходят из принципа, что у вас есть локальности обращений. Когда вы используете страницы, вы используете страницу один, а также шестнадцать или двадцать последующих страниц.

Для операционных систем это оказывается не так. Операционная система — та же самая программа. Данные, код, стек — чего там нового можно придумать? Однако обращения не оказываются локальными. Мы пробовали за раз подкачивать в операционной системе по два мегабайта, по шестнадцать страниц, по 64 килобайта, по сто двадцать восемь. Для старых операционных систем результаты были неплохие, а для новых это уже не работает. Пока мы остановились на варианте от одной до четырех страниц, в зависимости от разных операционных систем.

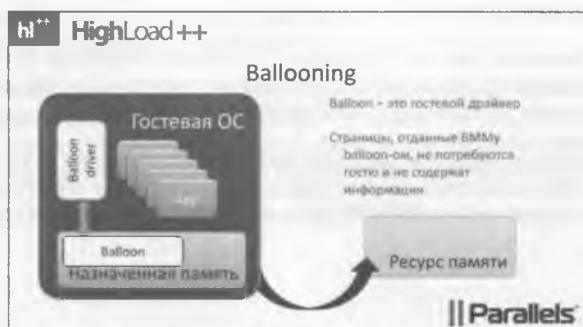
Есть одно преимущество, которым обладает виртуальная машина и которого нет у операционной системы. Мы можем внедрить своего агента в гостевую операционную систему. Отметим: операционная система страдать от этого не будет.

Назовем этого агента balloon. Balloon — это воздушный шарик...



Office-битва — это наш внутренний тест. Он очень простой, но крайне эффективный. Что он делает? Мы где-то поднимаем ноды и начинаем на них выполнять жуткие офисные операции. Мы открываем документы, закрываем документы, строим графики, делаем какую-нибудь статистику. Каждый раз мы отмечаем, сколько времени это занимает. Получается статистика по latency.

На этом сравнении (мы внутри себя решали, какие технологии мы будем использовать) видно, что этот таинственный ballooning очень сильно выигрывает по сравнению с вытеснением.



Что же такое ballooning? У вас есть назначенная память. Она очень плотная. В ней хранятся данные. Вы в нее вставляете маленький воздушный шарик и начинаете его надувать. Плотность данных увеличивается, но в шарике воздух, там пусто.

Ballooning — это такая дырка, которая оказывается в назначеннной памяти. Вы можете все страницы, которые оказались внутри этой дырки, безопасно отдать другой виртуальной машине.

Как это технически реализовано? У вас есть гостевая операционная система, есть шпион, который называется воздушным шариком — balloon. Он просто забирает у операционной системы немножко памяти. Он обычный драйвер, ни в чем не повинный. Он совершенно легально просит себе памяти.

Эту память он через tool gate — тайный обходной маршрут — отдает монитору виртуальной машины. Все легально, все довольны.

Чем довольны мы? Мы можем вытеснить память, которую никогда никто ни за что не будет использовать. С точки зрения гостевой операционной системы память уже оказалась занята.

Вопрос из зала: А что, если упадет? Ведь бывают ошибки в программе.

Анна Воробьевна: В программе бывают ошибки. У нас ошибок не бывает. (Смех в зале).

hl++ HighLoad++	
Ballooning	
Плюс	Минусы
<ul style="list-style-type: none"> ✓ Сокращение подкачки между ВММ и гостем 	<ul style="list-style-type: none"> ✓ Гостевой своплинг вплоть до гостевых крешей ✓ Неуниверсальность ✓ Отсутствие гарантий
 Parallels	

Balloon настолько хорош, что есть целые статьи «Overcommit и balloon. Нужно ли что-то еще?». Такие замечательные названия. Но, как правильно сейчас сказали, одна из основных проблем в том, что мы не знаем, насколько раздуть этот balloon. Balloon — это потребитель памяти, он может вызвать out of memory.

У нас случалось, что Windows crash'илась. Я даже не знала, что Windows может crash'ится от того, что у нее слишком мало памяти. Я думала, она просто вернет ENOMEM, и все будут довольны. Но она crash'ится, bsod'ит.

Неуниверсальность. Решение отлично подходит для Вашей Windows 8 и даже для Linux. Для Linux balloon вообще находится в основном дереве исходников. Однако для OS/2 или Windows 95 balloon не надуете никак. В Mac OS с balloon тоже сложно.

Наконец, отсутствие всяких гарантий. Мы говорим: «Выставляем квоту. Хотим через balloon съесть 512 мегабайт». Все классно, но у операционной системы внутри живет прожорливый sql-сервер. Он вам не отдаст 512 мегабайт. Хоть вы умрите, но он не отдаст.

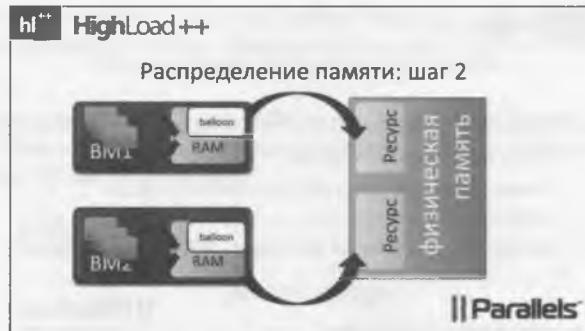
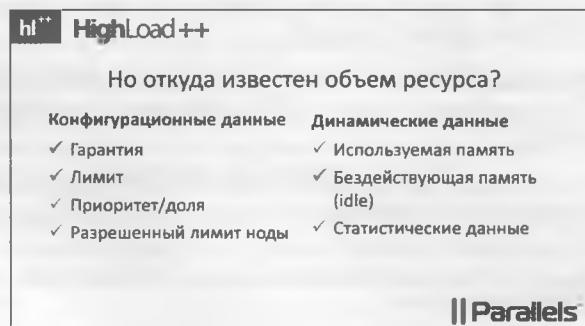


Схема меняется. На ней добавился balloon. Вот у нас появился внутригостевой swapping. У нас получилось два уровня swapping'a. Один — между виртуальной машиной и host'ом, а второй — внутри гостя, между гостевыми приложениями и назначенней памятью.

На этой схеме почти все хорошо. Мы уже отнимаем страницы, может вытеснить их, заметно или незаметно для гостя.

Дело в том, что, согласно этой схеме, получается, что ресурсов для всех виртуальных машин одинаковое количество. Мы с самого начала говорили, что это неэффективно. Мы хотим воспользоваться экспертым мнением уважаемого господина админа, который будет знать, какая из виртуальных машин для него самая любимая.



Как же это делается? У нас есть задаваемые, конфигурационные данные и те данные, которые мы получаем динамически.

Конфигурационные данные разнятся от одного продукта к другому. В целом это верхний и нижний пределы, какая-то доля, то есть приоритет этой виртуальной машины среди других виртуальных машин. У «VMware» это называется «налог на лень».

Есть еще разрешенный лимит ноды, то есть суммарный объем памяти, который можно потреблять всем виртуальным машинам. Почему это важно? Если у вас на ноде только виртуальные машины, то все отлично. Если у вас там заведен хотя бы один пользователь, то он может устроить там что-нибудь совершенно непристойное, и ваши виртуальные машины будут страдать.

hi HighLoad++**

Опасности конфигурируемых данных

- ✓ Избыток назначеннной памяти (32 по-пае + 4GB RAM)
- ✓ Своп из-за низкой гарантии
- ✓ Незаслуженный дефицит при лимите меньше назначеннной памяти
- ✓ Оптимистичный лимит для ноды

|| Parallels

Дальше стоит сказать о том, о чём говорят почти все мануалы. Каждый виртуализационный вендер выпускает мануал под названием «Best practises». Там обязательно есть раздел *memory management* и рекомендации по поводу того, что стоит делать, чего не стоит делать, какие есть сценарии. Я расскажу только о самых неосвященных на мой взгляд сценариях.

Первый: избыток назначеннной памяти. Звучит очень смешно, но есть реальные люди, коммерческие customer'ы, которые назначают 32-битной Windows XP восемь гигабайт памяти. Она не PAE, она никогда не увидит эти восемь гигабайт. Вы ей их назначили, вы ей отдали этот ресурс, а она, несчастная, видит только три.

Второй сценарий: swap из-за низкой гарантии. Вы поставили низкую гарантию. Эта гарантия не покрывает даже ядро операционной системы вместе с вашими любимыми приложениями. Тогда в случае overcommit'a вы можете страдать.

Некоторые особо выдающиеся личности могут установить маленький лимит. Лимит меньше назначеннной памяти. Это приведет к тому, что BMM'ный swap будет даже в том случае, если на host'e достаточно места.

Оптимистичный лимит для ноды. Вы слишком много отдали виртуальным машинам. Пользователь хочет запустить какое-то приложение, но не может этого сделать, и вся система зависает.

hi HighLoad++**

Распределение памяти: шаг 3

|| Parallels

Дополняем картинку. Появляется гипервизор, который распределяет ресурсы в соответствии с какими-то известными ему конфигурациями, статистическими данными. Появляются приложения, появляется swap.

В чём тут неувязка? Об этом в мануалах почему-то не пишут, и статьи от «VMware» и «Хеп» не сфокусированы на этом. Куда дальше у нас будут попадать данные? Есть два гигабайта

данных. Часть их лежит в памяти. Куда мы денем полтора гигабайта? Для этого нам обязательно нужен *backing storage*, то есть какое-то хранилище, где мы будем хранить все неиспользуемые данные.

Почему важно говорить о *backing storage*? Если вы не фанатики, если вы не любите до конца оптимизировать, вам это не важно. Если вы действительно озабочены производительностью, вы будете думать о *backing storage*.

Backing storage

Влияет на	Популярные решения
✓ Suspend/snapshot	✓ File mapping
✓ Resume/switch to snapshot	✓ Anonymous mapping
✓ Подкачка	✓ HugeTLBs

|| Parallels®

Backing storage влияет на скорость подкачки, скорость сброса, suspend, snapshot, resume, switch to snapshot. Все операции, которые провоцируют активную работу с backing storage, могут очень сильно страдать при выборе file mapping.

Что такое *backing storage*? Это то, на что мы делает map. File mapping означает, что все данные будут отображаться напрямую из памяти процессов в файл на диске. Всю работу по синхронизации между этими двумя storage берет на себя операционная система.

В чем недостаток? Это достаточно медленные операции. Закрытие будет очень быстрым. Вы закрыли приложение, а операционная система сама в фоне сможет все записать, но целом операции достаточно медленные. Все-таки работа с диском.

Есть анонимный mapping. Чем он хорош? Он хорош достаточно быстро работой, потому что swap может быть отдельным разделом на диске, файлом. Кроме того, вы можете туда встроить ваши механизмы. Вы сами должны будете синхронизировать данные, вы сможете сделать с ними все, что угодно.

Есть такая классная штука, как использование файловой системы Huge TLBFs, то есть вы используете большие страницы.

В чем идея больших страниц памяти? Размер страницы — это фрагментарность работы с памятью. Если вы используете маленькие страницы (4 кило-байта), то можете позволить себе меньшую локальность обращений, но ожидайте больше страницных промахов. Если вы используете большие страницы, то у вас меньше промахов, но при нелокальности обращений существенная доля памяти подмаппирована за зря.

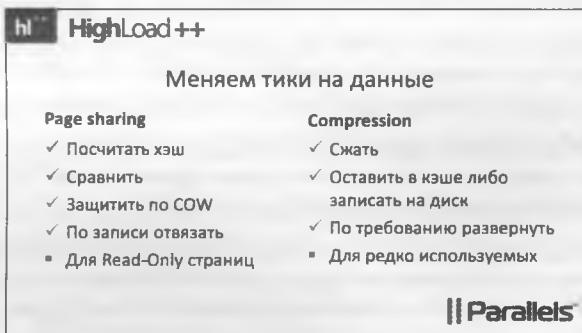
Преимущество Huge TLBFs в том, что она быстрее, потому что промахов меньше. Минус в том, что overcommit с ней по-настоящему не работает. Я об этом скажу.



Так это выглядит на схеме. ВП — это виртуальная память. Я специально обозначала ее как ВП, а не VM (virtual memory), потому что иначе бы получилась путаница с ВМ (виртуальная машина). Здесь же backing storage.

Почти все хорошо. Я уже подхожу к логическому концу рассказа о технологиях.

Что еще здесь можно сделать? Когда происходит запись в backing storage, запись все равно медленная, процессор простояивает, диск работает. Мы готовы на все. Мы готовы приложить какие угодно усилия, чтобы обменять ненужные нам процессорные тики на скорость диска. Для этого есть целых два решения.



Первое — page sharing — разделение страниц, а второе — компрессия. Эти технологии являются такими же старыми, как и виртуализация.

Я видела мужчину, который долго говорил об алгоритмах компрессии в условиях, когда оперативная память была очень дорогая. Он говорил, что пока это плохо реализуемая технология, потому что процессоры слишком медленные. Он говорил, что, когда деревья будут большими, в каком-нибудь 2012 году, можно будет наконец использовать компрессию. Это свершилось, потому что в Linux KSM и Transparent Page Sharing уже встроены.

Что такое компрессия и page sharing? Page sharing: мы пытаемся использовать одну копию идентичных страниц. Когда это нужно? Вы используете однотипные виртуальные машины. Другой вариант: у вас несколько процессов в системе (я говорю про Linux) используют одну статическую библиотеку. Ее можно разделить. Нулевые страницы отлично share'ятся.

Как происходит page sharing? Мы считаем хэш для каждой страницы, сравниваем с имеющимися и если находим дубликат, то устанавливаем отображение проверенной страницы на физический дубликат, выбрасывая реальную страницу. Если страница не Read-Only, то мы защищаем ее от записи и отвяжем отображение в случае записи в нее (Copy-On-Write).

Компрессия: мы просто сжимаем страницу и дальше можем оставить ее в кэше, чтобы при удобном случае записать. По требованию мы ее разворачиваем.

В чем недостатки обеих технологий?

hi++ HighLoad++

Меняем типы на данные

Page sharing	Compression
✓ Эффективность при однотипной нагрузке на ноду	✓ Интеграция с suspended image
■ Накладные расходы могут быть напрасны	✓ Уверенный compression вне зависимости от нагрузки
■ Запрет на большие страницы	

|| Parallels

Page sharing замечателен, изумителен, очень хорош при однотипной нагрузке на ноду, но накладные расходы могут быть напрасными. Если вы админ, вы можете сконфигурировать однотипные и только однотипные виртуальные машины. В этом случае можно использовать page sharing. В других случаях это может быть избытком. Каким бы быстрым ни был процессор, подсчет хэша потребует много ресурсов. Кроме того, страницу из памяти надо подкачать, пройтись по ней.

Page sharing — это запрет на большие страницы. Почему? Вероятность того, что 2 мегабайта окажутся идентичными, куда меньше вероятности того, что 4 килобайта окажутся идентичными.

Чем хорош compression? Он дает уверенный прирост производительности, вне зависимости от типа нагрузки. Кроме того, иногда нам требуется сохранить всю память на жесткий диск или передать каким-то образом по сети в рамках живой миграции (live migration), и тогда компрессию можно интегрировать. Она нам очень удачно поможет.



У нас получается полная картинка. Вот так все это работает. У нас есть balloon, который надувается, страницы, которые вытесняются, гипервизор, который каким-то образом распределяет ресурсы между виртуальными машинами. На все это навешены архивирование, компрессирование и backing storage.

У кого-нибудь есть вопросы по этой картинке?

Вопрос из зала: В нормальной жизни balloon будет вытесняться самой операционной системой. Как вы с этим боретесь?

Анна Воробьева: Что вы имеете ввиду?

Вопрос из зала: Вы выделили кусок памяти, память не используется, balloon будет вытесняться.

Анна Воробьева: Память с точки зрения операционной системы используется. В том-то и дело.

Вопрос из зала: Она выделена. Что угодно будет идти по памяти...

Анна Воробьева: У вас есть какой-то драйвер, который залочил память. Операционная система не знает, что этот драйвер — balloon. Он аллоцирует page bool, то есть невытесняемые страницы.

Вопрос из зала: Для разных операционных систем вы по-разному реализуете алгоритм...?

Анна Воробьева: Нет, не по-разному. В Linux balloon — это PCI-устройство, там все...

Вопрос из зала: По-разному все-таки. Реализуете его в разных операционных системах, чтобы он был невытесняемым участком...

Анна Воробьева: Да, он всегда невытесняемый. Есть еще вопросы?

Вопрос из зала: Пользуясь вашей терминологией, у нас есть воздушный шарик. Это драйвер или устройство, которое каким-либо образом может взаимодействовать с host'овой операционной системой и гипервизором...

Анна Воробьева: С гостевой. Он через tool gate ходит к монитору виртуальной машины — гипервизору.

Вопрос из зала: В том-то и дело. Получается, если использовать вашу терминологию, это агент. Агент может быть двойным. Проблема вот в чем: не может ли он создать дырок в безопасности таким образом, чтобы с помощью этого воздушного шарика две виртуальные машины могли получить доступ к пространству друг друга?

Анна Воробьева: Нет. Balloon передает только физические индексы страниц, которые он вытеснил. Он входит через tool gate. Мы анализируем запросы и отсеиваем те из них, которые не подлежат исполнению.

Вопрос из зала: Правильно ли я понимаю, что balloon реально в памяти машины не существует? Эта страница, которую вытеснили в кэш.

Анна Воробьева: Да.

Вопрос из зала: Host-система знает о том, что эта страница не используемая...

Анна Воробьева: Она не знает. С ее точки зрения наш гипервизор забрал страницы и как-то начинает их распределять. Когда мы получили страницы от balloon'a, мы их спокойно отдаем другой виртуальной машине.

Вопрос из зала: В обратную сторону работает?

Анна Воробьева: Я забыла об этом упомянуть, но это одна из самых сложных штук. Забрать память легко, а вот отдать ее очень сложно.

Вопрос из зала: Что происходит в Linux, если я нахожусь на гостевой машине, вижу свободную память, но при этом у меня по нулям буфера и кэш? Что с этим в целом происходит? Я вижу всю выделенную память, она вся моя для всех процессов, но на самом деле кэш все равно работает и буфер есть.

Анна Воробьева: Не совсем поняла вопрос. В чем проблема?

Вопрос из зала: Я на своей машине вижу два гигабайта памяти, которые выделены. При этом кэш и буферы стоят по нулям, потому что я в виртуальной машине.

Анна Воробьева: Потому что balloon все съел, например.

Вопрос из зала: Я не знаю, есть ли там balloon. Я говорю с точки зрения конечного пользователя. Я вижу, что кэш в файлах все равно работает в Linux...

Анна Воробьева: В гостевом или host'овом?

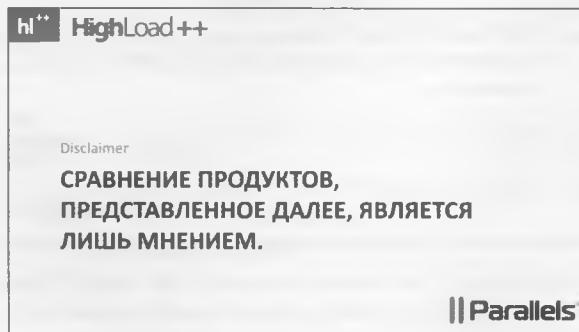
Вопрос из зала: В гостевом. Как распределяется память на кэш среди всей физической машины?

Анна Воробьева: Имеется ввиду — виртуальной машины? Давайте мы с вами чуть попозже обсудим это, я попытаюсь более четко вникнуть в ваш вопрос. Извините, пожалуйста, за нехватку внимания.

Вопрос из зала: (Неразборчиво, тихо, 36:55 — 37:00).

Анна Воробьева: Нет, мы говорим о виртуальных машинах уровня окружения. Я с самого начала это вводила. Контейнер — немножко другая вещь.

То, о чём просили организаторы «HighLoad», и то, что я попытаюсь сейчас вам дать.



Дальше я попытаюсь описать самые популярные продукты серверной виртуализации с точки зрения их фич по памяти. Предупреждаю, что это всего лишь мнение. Понятно, что исходники ESX мне недоступны, а исходники Хеп настолько запутаны по сравнению с KVM или VirtualBox, что... Я надеюсь, что я их поняла.

The image shows a slide from a presentation titled "HighLoad++". At the top left is the logo "Hl++ HighLoad++". Below it is a table comparing memory features across different hypervisors. The table has columns for Balloon, Page sharing, Compression, Swapout, Quota, and Backing store. The rows list ESX, Xen Server, P5BM, HyperV, and KVM. The table shows varying levels of support (+ or -) for each feature across the listed hypervisors. At the bottom right of the slide is the Parallels logo.

	Balloon	Page sharing	Compression	Swapout	Quota	Backing store
ESX	+	+	+	+	+	Huge/VSWP?
Xen Server	+	-	-	-	+	Own
P5BM	+	-	+	+	+	Huge/File/An
HyperV	+	-	-	-	+	File
KVM	+	+	-	+	+	Huge/File/An

Первая сводная табличка. С одной стороны перечислены используемые механизмы, а в колонках представлены продукты. Я рассматривала HyperV, KVM, PSBM и ESX. PSBM — это Parallels Server Bare Metal, я о нем чуть позже скажу, это наша новая серверная виртуализация.

ESX поддерживает все. Они действительно молодцы. Вальдспургер (Waldspurger) в 2001 году написал статью по поводу работы с памятью. Уже в этой статье было покрыто подавляющее большинство современных «фич». Они придумали balloon. Они используют Huge страниц в качестве backing storage и специфическую свою разработку VSWP, которая скорее относится к анонимному mapping'у.

В Xen Server — я говорю про Xen Server, я не говорю про Xen — отсутствуют почти все фичи. Они крайне осторожные ребята. Они используют собственную имплементацию backing storage.

В PSBM мы пока не поддерживали page sharing, но поддерживаем все backing storage. Все остальное тоже у нас хорошо.

Hyper-V — особый разговор, я о них расскажу.

KVM не поддерживает только компрессию, а так у них полный набор функционала.

Теперь подробнее по продуктам.

hl++ HighLoad++

Сравнение: Xen Server – осторожность превыше всего

- ✓ Исключительно ballooning
- ✓ Page-sharing & swapout присутствуют в xen hypervisor 4.0

|| Parallels

Xen Server, коммерческий продукт, в отличии от нормального Xen... Прошу прощения, от open-source Xen... (Смех в зале). Xen Server имеет ограниченный функционал.

Они сказали: «Мы поддерживаем только ballooning». Они не хотят ни swapout страницы, ни делать page sharing. Page sharing был внедрен в Xen уже в 2010 году. Почему не хотят? Это политика безопасности. Другого я предположить не могу.

Когда это будет внедрено, сказать не могу. Там, по-моему, все не очень весело у этих ребят. Но, судя по тому, что говорят друзья-linuxoid'ы, Xen не очень прогрессирует в последнее время.

hl++ HighLoad++

Сравнение: VMWare ESX – сильнейшие со времен Waldspurger-a

- ✓ В статье 2002ого года они уже описывают balloon, kvotu, page sharing, idle-memory tax swapout
- ✓ Некоторая инертность в новом compression не интегрирован с suspend-ом

|| Parallels

«VMware». Как я говорила уже, в 2002 году они реализовали практически все. Но сейчас у них начался некий простой. Недавно прикрутили компрессию (это была единственная «фича», которую не реализовали в 2002 году), но не смогли ее интегрировать с suspend. Не знаю, с чем это связано. Ребята говорят, что там стало скучно, и просятся к нам.

hi++ **HighLoad++**

Сравнение: KVM – все блага Linux-а

- ✓ Balloon включен в дерево Linux
- ✓ Эффективнейший KSM достался бесплатно
- ✓ Блага надежного вытеснения
- ✓ Compression и алгоритмы, специфичные для виртуализации, могут идти с запозданием

|| Parallels

KVM. Они очень большие молодцы. Они все получили на халяву. Таких халявщиков еще поискать надо. Page sharing они получили от ядра, balloon – в дереве Linux, все вытеснения за них делает Linux. Им вообще ничего делать не надо. Они маленькие, аккуратненькие, шустрые.

Проблема в том, что в сообществе специфические для виртуальных машин алгоритмы пока что не делают. Я не знаю, когда их будут делать. Глаубер Коста, разработчик KVM, говорит, что ресурсов нет, он один закрывает собой амбразуру. Поэтому я не знаю, когда они реализуют компрессию.

hi++ **HighLoad++**

Сравнение: HyperV – все что не от нас, то от лукавого

- ✓ Hot-plug memory + balloon
- ✓ Overcommit опасен и вреден

|| Parallels

Hyper-V. Это классика жанра. Я никого не хочу обижать, но когда я читала последний пресс-релиз, я просто плакала. Эти ребята на полном серьезе говорят, что overcommit – это очень вредно. «Microsoft» его не поддерживает, а все остальные компании, которые поддерживают overcommit, врут customer'ам. Там было сказано: «Мы не хотим обманывать наших customer'ов, мы не используем overcommit».

У них есть так называемая startup-memory. Это то, что с самого начала выдается гостю. Затем она наращивается при помощи hot-plug memory. Можно вставить карту памяти, и гостевая операционная система ее увидит. Они через hot-plug memory докидывают туда памяти.

Если нужно отобрать память – не из-за overcommit'a, а просто нужно отобрать, – то они надувают balloon. Как они его сдувают, там не говорится. У меня есть подозрения, что они эту проблему не решили.

О нас любимых. У нас сделали классный алгоритм компрессии. Его делал очень мозговитый парень. Он получил очень хорошие результаты, которые действительно работают.

hl++ HighLoad++

Сравнение: PSBM

- ✓ Свой алгоритм компрессии и его интеграция:
 - ✓ Эффективная реализация для разнотипной нагрузки
 - ✓ Быстрый suspend/resume/snapshot
- ✓ Для однотипной нагрузки – контейнеры

|| Parallels

Для однотипной нагрузки (у нас нет page sharing) мы в PSBM предлагаем использовать контейнер. PSBM – это такой гибрид. Там есть и контейнерная виртуализация (наша Virtuozzo Linux), и гипервизорная виртуализация, использующая тот же движок, что и Parallels Desktop.

hl++ HighLoad++

	Balloon	Page sharing	Cmprs	Swap	Quota	Backing store	Usage
ESX	+	+	+	+	+	Huge/VSSP?	Hi-end ENT
Xen Server	+	-	-	-	+	Own	Providers, middle Linux ENT
PSBM	-	-	-	+	+	Huge/File/An	Service providers, ENT
HyperV	+	-	-	-	+	File	Windows middle ENT
KVM	+	-	-	+	+	Huge/File/An	Providers

Просили сказать, где и что лучше использовать.

VMware ESX – это, без сомнений, лидер high-end enterprise'a. Xen Server используется провайдерами и теми видами enterprise-бизнеса, где не нужны high-end «фиши». На мой взгляд, KVM вскоре заменит Xen, потому что на его стороне поддержка всего сообщества, а Xen ее постепенно лишается. Hyper-V – отличный продукт в случае, если вам нужен только Windows. PSBM. Мы исторически работаем с сервис-провайдерами, но enterprise-бизнес у нас тоже растет и улучшается.

Такая получилась схемка. Теперь я хочу спросить, есть ли у вас вопросы? Спасибо всем тем, кто дослушал этот доклад до конца.



Вопросы и Ответы

Вопрос из зала: Здравствуйте. У меня к вам смежный вопрос, поскольку вы много участвуете в разработке контейнерной виртуализации, особенно LXC. Стоит ли обращать внимание на гипервизорную виртуализацию, если в моем конкретном случае подходит контейнерная?

Анна Воробьева: Если вам нужна только контейнерная виртуализация...

Вопрос из зала: Мне нужно виртуализовать n-одинаковых машин. Стоит ли вообще заморачиваться...?

Анна Воробьева: Контейнерная. Однозначно.

Вопрос из зала: Второй
Вопрос: сделаете в LXC reboot?

Анна Воробьева: Что сделаю?

Вопрос из зала: В Linux host containers вы половину написали, а reboot'а там до сих пор нет. В OpenVZ есть. В LXC очень этого не хватает.

Анна Воробьева: Я передам нашим ребятам, пусть сделают.

Вопрос из зала: Вы рассказывали про компрессию в памяти. Нужна ли она? Кто реально ее может использовать с пользой для себя? Уровень сжатия там наверняка будет довольно небольшой, а время разжатия и затраты, связанные с этим, довольно большие? И проводили ли вы какие-то...?

Анна Воробьева: Мы проводили большое количество тестов, прежде чем решили использовать компрессию вообще и выбрали этот

частный алгоритм. Мы очень аккуратно относимся к performance. Зачастую мы делаем какое-то решение, проверяем, и если прирост производительности недостаточно велик, то мы отказываемся от внедрения. Мы сравниваем не только с накладными расходами, но и с той опасностью, которую технология может принести (какие-то пики нагрузки или ошибки).

Компрессию мы тщательно проверяли. Она действительно дает прирост. Она дает прирост просто из-за того, что диски слишком медленные. Даже по сравнению с SSD. Если диски ускорятся, то компрессию можно будет опять выкинуть.

Вопрос из зала: Сколько занимает по времени распаковка страницы?

Анна Воробьева: По сравнению с чем?

Вопрос из зала: В абсолютном времени. Время поиска страницы на диске понятно какое, все знают. Каков порядок конкретно здесь?

Анна Воробьева: Давайте обменяемся с вами контактными данными, я вам скажу. У нас есть тестовые данные. Я не помню чисел.

Вопрос из зала: Можно пару слов про сдувание balloon'a? Как он надувается понятно.

Анна Воробьева: Чисто технически мы говорим balloon'у через tool gate: «Верни, пожалуйста, страницы, 15 страниц». Balloon возвращает их. Просто делает возврат на host этих страниц, освобождает их, release'ит.

Проблема в том, когда это делать. Вроде у нас все устаканилось. Мы видим какую-то возрастающую нагрузку и говорим: «Раз устаканилось, давайте вернем страницы». В момент, когда мы возвращаем страницы гостевой операционной системе, она может начать разбалансировать нагрузку. Обратно забирать у нее получается очень дорого, поэтому это очень тонкий момент.

Мы это делаем через специальный Idle Memory Thread. У нас есть thread, который отслеживает уровень idle'ности системы. Таким образом решаем вопрос о возвращении страниц balloon'.

Вопрос из зала: Сдуть в обратную сторону его можно? Надуть в минусовую часть?

Анна Воробьева: Вы хотите сделать так, чтобы balloon был таким выпуклым прикатком, аппендиксом из...? (**Аплодисменты**). Наверно эта технология называется memory hot-plug.

Вопрос из зала: Да, именно так.

Анна Воробьева: Hot-plug у нас поддерживается.

Вопрос из зала: На основе balloon?

Анна Воробьева: Нет, это должна делать операционная система. Представьте ужас операционной системы. У нее появляется программа, которая говорит: «Теперь магическим жестом у тебя памяти станет в два раза больше».

Вопрос из зала: У меня есть любимая виртуальная машина — qemu. Почему про нее ничего не сказали?

Анна Воробьева: Про qemu? Про коммерческие решения на основе qemu? Дело в том, что qemu очень медленный.

Qemu в действительности отличный движок, как и Xen, и KVM. KVM как таковой — это маленький wrapper между Linux и самой hardware-виртуализацией. Они используют устройства qemu. Они используют qemu для каких-то взаимодействия, для командой строки.

Сам по себе qemu достаточно медленный. Он user-space'ный, он не может поддерживать hardware-виртуализацию, а это очень выгодно.

Вопрос из зала: Почему не рассмотрели OpenVZ?

Анна Воробьева: OpenVZ — это контейнерная виртуализация.

Можно виртуализовать на трех уровнях. Можно виртуализовать на уровне приложений — это виртуальная машина Java. Можно виртуализовать на уровне операционной системы — это Hyper-V, OpenVZ. Наконец, можно виртуализовать на уровне hardware — это VMware ESX.

Вопрос из зала: Про balloon еще вопрос. Кто контролирует ту память, которую он запрашивает у гостевой операционной системы?

Анна Воробьева: Гостевая операционная система ее теряет. Монитор виртуальной машины отдает эту память назад в гипервизор, а гипервизор уже решает, что с ней сделать.

Вопрос из зала: Кто определяет объем забираемой памяти?

Анна Воробьева: Гипервизор.

Вопрос из зала: На основании чего он берет данные?

Анна Воробьева: Уровень нагрузки системы, уровень уже залоченных balloon'ом страниц (чтобы не случилось ситуации с Оом-киллером), уровень компрессии. Очень много факторов, которые в действительности влияют. У нас есть специальная настройка конфигурации, которая задает максимальный уровень надутия balloon'a.

Мы получили нехватку страниц. Нам из-за чего-то нужно у этой виртуальной машины вытеснить страницы. По каким технологиям вытеснять — решает гипервизор.

Вопрос из зала: Про миграцию еще расскажите, пожалуйста, немножко.

Анна Воробьева: Что вы хотите узнать о миграции?

Вопрос из зала: Как она выглядит.

Анна Воробьева: Миграция — это очень интересная тема. Если бы мы остановили виртуальную машину и мигрировали бы ее на host, мы были бы ограничены только сетевым трафиком. Взяли данные и перекинули — все отлично. Мы не останавливаем. Мы говорим про live-миграцию. Проблема в том, что мы уже перевели часть памяти, а гость все продолжает дописывать.

Люди из MIT (Массачусетский технологический институт) доказывают, какой алгоритм эффективнее, показывают разные математические модели на эту тему. При миграции нам нужно передавать и диск, и память. Так как мы swap'им, то в действительности обновляемые данные на диске являются зачастую дубликатом того, что находится в памяти. Если их смотреть, то можно передавать в два раза меньше данных.

Вообще это очень сложная тема. Она на час. Я понимаю, что времени много, но...

Вопрос из зала: Могли бы вы рассказать в двух словах? Насколько начинает вратить scheduler гипервизора при интенсивной работе с памятью?

Анна Воробьева: Вратить? Мы не врем.

Вопрос из зала: Производится большой объем перекидки памяти. Как сильно тот квант времени, который уходит на гостя, будет...?

Анна Воробьева: Вы имеете ввиду, насколько будет тормозить время в госте при большой нагрузке на память?

Вопрос из зала: Я хочу понять, насколько система начинает быть нечестной. У меня очень много real-time процессов. Я пытаюсь понять, можно ли вообще использовать хоть какую-то виртуализацию.

Анна Воробьева: Я бы сказала, что, если можно, то это контейнерная виртуализация. У них overhead меньше. У вас soft real-time или hard real-time?

Вопрос из зала: У меня real-time видеопроцессинг. Soft достаточно...

Анна Воробьева: Если soft, тогда вам можно использовать виртуализацию. Нужно будет правильно выставлять квоты и следить за количеством ресурса. Главное — использовать специальные hardware-«фичи» по таймерам, и все у вас будет хорошо.

Вопрос из зала: Мы здесь собирались на конференции «HighLoad». Это в основном серверные технологии. Все наши приложения — MySQL, Redis, MongoDB — аллоцируют память и потом ее не отдают. Я запускаю MySQL, он аллоцирует всю память для того, чтобы она не фрагментировалась, чтобы локальность была правильной. Какой смысл в overcommit, компрессии и так далее, если у нас все запускается, аллоцировав всю память?

Анна Воробьева: Как он аллоцирует память? Он ее лочит?

Вопрос из зала: Нет, не лочит. А как она перераспределится?

Анна Воробьева: Вы думаете, он ее использует целиком?

Вопрос из зала: Да, конечно. MySQL, набрав кэш, начинает использовать ее целиком. Когда он запускается, он просто ее аллоцирует. Потом он начинает набирать буфер.

Анна Воробьева: Спрашиваете, зачем тогда использовать?

Вопрос из зала: Да. Я не вижу смысла использования. Если MySQL не набрал кэш, то он практически не работает. Он работает только тогда, когда все данные находятся в памяти. Java тоже аллоцирует всю память для своего heap'a. Если у нее в heap ничего нет, то она очень медленная.

Анна Воробьева: С их точки зрения все будет в heap, все будет в их кэше. Это с их точки зрения, а вы можете этим вертеть как хотите.

Вопрос из зала: Приложение не знает этого. Из-за того, что у него реализован свой механизм работы с памятью, оно может думать, что это действительно находится в оперативной памяти.

Анна Воробьева: Мы внедрим какую-нибудь компрессию или что-нибудь такое. В итоге это может оказаться эффективным. Надо померить. Без тестов я боюсь говорить.

Вопрос из зала: У вас есть такие тесты?

Анна Воробьева: Есть наверняка. В недрах компании у нас безумное количество тестов. Я недавно обнаружила, что мы нормально тестируем power setting. Я думала, что у нас этого нет.

Интересный момент. Exchange с четырьмя процессорами работает не так хорошо, как с двумя. Две виртуальные машины, в каждой из которых по два процессора и запущен Exchange, работают быстрее, чем Exchange с четырьмя процессорами. Такое забавное наблюдение по поводу процессоров. Возможно, что с памятью то же самое.

Вопрос из зала: Я понимаю, зачем разделять на процессы. Было время, когда MySQL плохо работал с количеством ядер больше восьми.

Анна Воробьева: Я просто говорю о том, что такое умышленное и преступное скрытие данных от приложения может дать хороший performance gain. Надо провести измерения, надо поискать эти тесты у нас в компании.

Вопрос из зала: Давайте рассмотрим следующую ситуацию. Имеем гостевую машину с надутым balloon'ом. Вдруг эта машина перезагружается. Как себя поведет гипервизор?

Анна Воробьева: От чего она может перезагрузиться?

Вопрос из зала: Из-за действий пользователя.

Анна Воробьева: Balloon, как любой драйвер, в этот момент застопится, потому что стопится сама гостевая операционная система. Он отдаст всю память. Вообще вся память будет от-

дана. При старте она аллоцируется, мы будем набирать новую статистику и работать, исходя из этой новой статистики. Возможно, будем надувать balloon, возможно, в нем необходимость уже пропала.

Вопрос из зала: А если загрузится другая операционная система? Я обновил ядро Linux, перезагрузил и получил совершенно другое ядро.

Анна Воробьева: Когда вы перезагрузитесь, вся история жизни вашей виртуальной машины начинает идти заново.

Вопрос из зала: Мы будем наблюдать нехватку памяти?

Анна Воробьева: Если на ходе есть нехватка памяти, то от того, что вы перезагрузите виртуальную машину, памяти больше не станет.

Вопрос из зала: Часть памяти уже съедена другой виртуальной машине?

Анна Воробьева: Назначена. У вас на машине два гигабайта памяти и две виртуальные машины, каждая по два гигабайта памяти. Нормальная ситуация. Одна работает, другая перезагрузилась. Когда она перезагружается, она эти два гигабайта будет себе набирать. У нас все равно будет состояние overcommit'a.

Вопрос из зала: Что начнет вытеснять гостевая операционная система в первую очередь, когда надувается balloon? Не получится ли, что она в первую очередь освободит ненужные на ее взгляд вещи вроде кэшей и буферов? Не останутся ли при этом неиспользуемые приложения в памяти? Нет ли способов более эффективно влиять на алгоритм поведения гостевой операционной системы, чтобы указать, что она в первую очередь будет вытесняться?

Анна Воробьева: Влиять на это возможности нет. Есть возможность мониторить. Это наше know how. Мы можем проверять, в каком состоянии находится гостевая операционная система. Если она будет вытесняться из кэшей, мы не захотим надувать balloon, потому что она от этого будет работать хуже.

Вопрос из зала: Вы говорите, что у вас есть memory hot-plug. CPU hot-plug тоже есть?

Анна Воробьева: Да.

Вопрос из зала: Могу я прорасыывать устройства типа PCI, USB и прочих внутри виртуальной машины?

Анна Воробьева: Да, без проблем.

Вопрос из зала: Могу я забирать эти устройства, даже если они нагружены?

Анна Воробьева: Про CPU unplug и memory unplug я не слышала. Насколько я знаю, ее просто физически не существует.

Вопрос из зала: Нет, почему же...

Анна Воробьева: Ее не поддерживают гостевые операционные системы. Для того, чтобы мы пробросили эту память, система должна ее увидеть. Мы на уровне ACPI-таблиц ей сообщаем, что у нее появилось больше памяти. Если мы заберем эту память, операционная система просто обидится и не захочет работать.

Вопрос из зала: Гипервизор Xen может легко забрать память и CPU, даже если они зарезервированы. Если памяти не будет хватать, то содержимое уйдет в swap. Если не будет хватать swap, сработаетoom-киллер. Система будет жить.

Анна Воробьева: Memory hot-plug делала не я, а другая команда. Возможно вы правы. Я про-верю. Спасибо.

Вопрос из зала: Вы все время говорите про hardware-виртуализацию. А что на счет паравир-туализации?

Анна Воробьева: Hyper-V, Xen — это паравиртуализация. KVM паравиртуализует некоторых гостей, даже мы немножко это делаем.

Вопрос из зала: KVM с этим плохо работает, Хен — хорошо. Хен может работать и в hardware-, и в паравиртуализации одновременно. У вас разговор сейчас идет полностью про hardware-виртуализацию. Какие у вас будут просесты по сети, если на одном каком-нибудь сетевом интерфейсе будет поднято много виртуальных машин?

Анна Воробьева: Вы имеете ввиду PSBM?

Вопрос из зала: Да.

Анна Воробьева: Вам прислать данные или вы хотите, чтобы я навскидку какие-нибудь цифры назвала?

Вопрос из зала: Можно прислать. Можно и навскидку сказать.

Анна Воробьева: Я вам пришлю лучше. Я цифры вообще не помню.

Спасибо вам большое. Спасибо, что выслушали.

Большая книга рецептов или часто задаваемые вопросы по управлению сложными системами

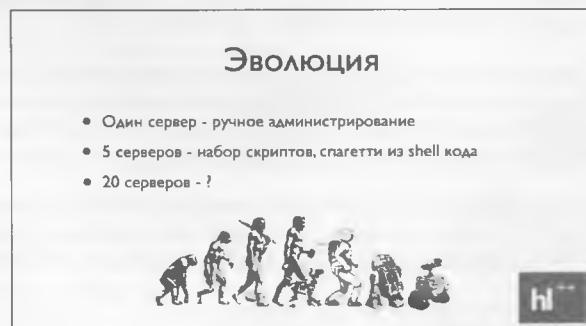
Александр Титов, Игорь Курочкин



Александр Титов: Добрый день! Все готовы? Я вижу, что все.

Меня зовут Титов Александр. Это мой коллега Игорь Курочкин. Мы вместе работаем в компании «Skype». Работаем системными администраторами и стараемся администрировать большую систему, по сути дела, в условиях стартапа. Мы работали в компании «Qik», который был стартапом и перешел в компанию «Skype». Сейчас мы находимся в процессе transition'a из маленького стартапа в большую компанию.

То, что мы будем рассказывать, — это не то, что используется повсеместно в «Skype». Это то, что используется в «Qik». Может быть, какие-то куски в дальнейшем будут использоваться и в «Skype».



Итак, немного про то, какие стадии мы переживаем, когда администрируем сервисы. Много-много лет назад — 5 лет назад (для меня это много) — практически у всех был только один сервер. Применялось ручное администрирование. Этот один сервер был основой всего во всех компаниях.

Существует даже притча, которая описывает эволюцию. Когда царь пришел к мудрецу и спросил его: «Как устроена Земля? Почему она не падает?», мудрец сказал: «Земля стоит на льве». Потом он спросил: «Хорошо. Почему лев не падает?» — «Лев стоит на слоне». — «Почему слон не падает?» — «Слон стоит на черепахе. Больше не спрашивайте меня, потому что дальше идут одни черепахи». Один сервер — это, по сути дела, оно и есть, основа всего.

С одним сервером все понятно. Он просто администрируется одним специалистом. Мы все знаем. Мы можем его затюнинговать. В принципе, мы можем разобраться с любой проблемой в какое-то минимальное время, потому что это все умещается в голове.

Дальше идет следующий уровень — 5 серверов. Те же самые подходы. Это уже более обобщается. Появляются наборы скриптов, кастомные, *in house*. Иногда это перерастает в спагетти из shell-кода на уровне больше пяти серверов (это тоже зависит от сервиса).

Когда 20 серверов, возникает вопрос.

Больше 20 машин - большая система

- Начинают возникать проблемы из-за разницы конфигураций
- Требуется большое количество документации
- Стоимость поддержки сильно превышает стоимость внесения изменения
- Надо управлять не отдельной машиной, а кластером



Для меня больше 20-ти машин — это, в первую очередь, большая система. Уже на этом уровне возникают проблемы из-за разницы конфигураций. Разные версии LIPC на одном сервере работают, на другом сервере не работают. Разные версии PHP, Ruby. Все это, я думаю, достаточно знакомо многим.

Начинает требоваться большое количество документации. Каждое изменение мы должны прописать. Если мы его не прописали — про него забыли. На конференции Яндекс месяц назад говорили, что они любят костыли, и костыли — это неотъемлемая часть системы. Да, это действительно так. Про костыли нужно записать: если его выставить, то все развалится. Про это часто забывают писать, особенно в условиях стартапа.

На количестве машин больше 20-ти стоимость поддержки сильно превышает стоимость внесения изменений. Мы перестаем вносить изменения, мы постоянно фиксим. Единственный способ избежать этого — в принципе, в большинстве компаний так и поступают — набирать больше людей, чтобы они работали в пределах их пяти серверов и не выходили за пределы этого.

Начинает приходить понимание, что надо управлять не одной машиной, а какой-то абстрагированной штукой — кластером. Это название очень любят менеджеры. «Почему у нас не работает?» — «У нас же кластер». Хочется управлять кластером (смеется). Правда, я такое слышал.

Чужой опыт

- Суперкомпьютеры
- Google
- Facebook
- Amazon



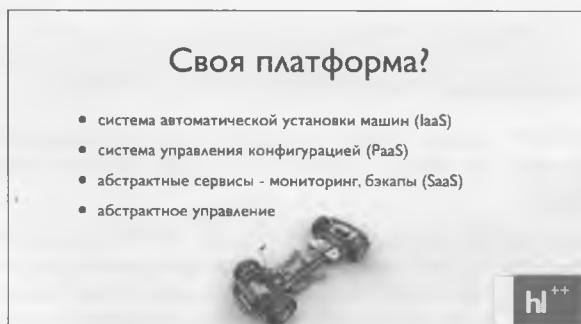
После этого логично обратиться к чужому опыту. Мы маленькие. Хочется узнать, как делают большие компании.

Первое, на что хочется посмотреть, — суперкомпьютеры. Много машин, большое количество вычислений. Они действительно должны быть одинаковыми по максимуму, иначе тоже не будет работать, а мы будем просто прожигать наши ядра.

Суперкомпьютеры действительно интересные. Есть Open Source версии системы управления компьютерами во Франции, в Германии. Есть их wiki, как они записывают, что они делают. Это все происходит в открытую. Это образовательные учреждения, они не получают денег, стараются все открыть по максимуму.

Но у них, в основном, продумана система автоматической загрузки всех машин. Они загружаются все с одним конфигом. Если им надо внести какое-то изменение, они запускают один скрипт на всех, производят это изменение. Если что-то сломалось, они просто переустанавливают машину заново. В принципе, достаточно общий подход.

Веб-компании используют несколько измененный подход. Google, Facebook, Amazon (это есть и в «Skype») в публикациях подходят к тому, что нужна некая платформа, на базе которой будет доступно абстрагирование от железа. Во-первых, это будет проще для разработчиков, потому что им даются какие-то конкретные «ручки», что надо дергать в этой платформе. Во-вторых, это проще для системных администраторов, потому что они, по сути дела, уже не администрируют отдельные сервисы. Они администрируют эту платформу, поддерживают ее работоспособность, а за конкретные сервисы (за неисправности) отвечают уже разработчики.



Итак, идея подходит к тому, что надо сделать свою платформу. В условиях минимальных ресурсов, когда я пришел в компанию «Qlik», мы решили сделать свою маленькую платформу, базирующуюся на Open Source. Нас было всего лишь три человека. И сейчас тоже три человека. Это было сделано за год.

Нам нужна система автоматической установки машин. В известной маркетинговой нотации это называется Infrastructure as a Service. Система управления конфигурацией. Мы будем называть это Platform as a Service, потому что система управления конфигурацией как раз создает те «ручки» для разработчиков и системных администраторов, которые позволяют управлять абстрагированно всей системой.

Абстрактные сервисы, которые тоже доступны через эту платформу как Software as a Service. Это мониторинг, бэкапы.

Абстрактное управление всем. Например, я хочу сделать изменение на всех веб back-end'ах. Я хочу сделать это, не запуская for... bla-bla-bla do ssh и так далее, а каким-то более простым, более понятным человеку способом.

Дальше про это пойдет речь. Сейчас Игорь расскажет про систему автоматической установки машин, которую мы используем.

Cobbler. Автоматическая установка машин

- Поддержка CentOS, RHEL, Debian, Ubuntu
- Физические и виртуальные машины
- Удобные инструменты - CLI, Web, API
- Передача сервера в Chef




Игорь Курочкин: Когда мы пришли в «Qik», сервера устанавливались вручную. Дата-центр в Америке, требовалось присутствие специалиста. Разные часовые пояса. В итоге новые сервера запускались долго, переустанавливались еще дольше. Конфигурация различалась. В общем, был бардак. Влиял человеческий фактор, поэтому управлять этим было невозможно.

Первое, что мы решили автоматизировать, — установка машин. Плюс мы хотели использовать виртуализацию для девелоперских окружений и инфраструктурных задач. Мы решили использовать уже готовые решения. Попробовали много вариантов, в итоге остановились на Cobbler. Почему?

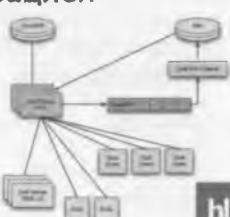
Так исторически сложилось, что на большинстве серверов у нас был CentOS, а виртуализацию мы лучше знали Хеп. Поэтому Cobbler нас полностью устроил.

Что умеет делать Cobbler? Cobbler умеет автоматически устанавливать как физические, так и виртуальные машины. Он имеет удобные инструменты для работы в консоли, веб-интерфейс, API.

Он скрывает от системного администратора настройку TFTP-сервера, DHCP-сервера, генерацию файлов ответов и так далее. Вы один раз импортируете дистрибутив, подключаете репозитории, создаете файл ответов, общий как для виртуальных машин, так и для физических серверов. На выходе у вас готов сервер, в котором созданы разделы, настроена сеть, подключены репозитории, отключены ненужные сервисы, установлены дополнительные пакеты. Самое главное — сервер передан под управление системы управления конфигурацией. В нашем случае это Chef.

Chef. Управление конфигурацией

- Ohai (база данных о хостах)
- Cookbooks
- Roles
- Environments




Александр Титов: Нам нужна была не просто система управления конфигурацией. Нам нужна была система, которая дает полное абстрагирование. Еще есть система управления конфигурацией Puppet. Он хорошо делает только одну эту функцию. Он только управляет конфигурацией.

Чем Chef отличается от Puppet. Он имеет еще базу данных, в которой хранятся вся информация обо всей системе. Эту информацию можно запрашивать через API с каждого конкретного хоста. База данных называется Ohai.

Также в Chef используются Cookbooks. Это сущность, которая описывает, что надо выполнить на сервере. Установка MySQL, прописывание конфигурации, изменение конфига, перезапуск сервиса и так далее. Из Cookbooks можно использовать всю ту информацию обо всех машинах, которые хранятся в базе данных Chef'a.

«Роли» (Roles) — это уже более абстрактное описание сервиса целиком. Например, если это сервис для e-mail рассылок, то в Roles описывается, как подключается и Postfix, и логирование, и все остальное. Самое главное — прописываются все переменные, которые требуются для отдельного сервиса на отдельной машине.

Environments — это классическое разделение на production, stage, development. При помощи Environments можно в Chef разделять куски системы по конфигурации.



Игорь Курочкин: Поиск в Chef — это его killer фича. Чем он лучше Puppet, например. Когда на какой-то машине проходит Chef Run, Ohai собирает все атрибуты машины. Они объединяются с атрибутами в рецептах, в ролях, в окружениях. Все эти атрибуты индексируются, и по ним можно вести поиск.

Что это значит. Это значит, что нам неважно, сколько у нас сейчас серверов. Мы всегда можем получить любой атрибут, скажем, IP-адрес. Мы можем спросить: «Дай мне список IP-адресов, веб back-end'ов на stage-окружении». Мы это получим. Мы не забиваем голову. Разработчикам тоже все равно, сколько там машин. Мы находим, и это работает.

Например, в базах данных slave находят master, клиенты находят сервер, front-end находят back-end, и наоборот.

Мониторинг



- Zabbix API
- Hosts, Groups, Templates
- Applications (Items, Triggers, Graphs)

hi⁺⁺

Теперь мы расскажем о том, что мы еще можем делать в Chef-рецептах. Например, мониторинг. В качестве мониторинга мы используем Zabbix, потому что у него есть API. Этот сложный API обернут в простой ресурс в Chef'e. Мы можем вызывать его как в рецептах, так и в консоли.

Что мы можем делать. Мы можем добавить машину в мониторинг. Мы можем добавить ее в группу. На машину можем добавить какие-то шаблоны с проверками. Но все эти шаблоны статические. Если на машине один сервис, все хорошо. Если на машине уже 20 инстансов Memcache или 64 инстанса Redis, то добавлять вручную эти проверки уже сложно, порой даже невозможно. Поддерживать это тоже никак нельзя.

Мы пошли дальше. Мы можем из рецепта создавать приложения с уникальными Items, Triggers, Graphs. Например, для статистики у нас 30 тысяч Items, 7 тысяч Triggers и всего 3 человека в отделе. Мы не создавали их руками. Все автоматизировано.

Бэкапы

- Конфигурация в Chef
- Бэкапим статику, БД и репозитории
- Автоматизируем

hi⁺⁺

То же самое касается бэкапов. Если вся конфигурация находится в Chef, то бэкапить конфигурацию не нужно. Мы бэкапим только статику, базу данных и репозитории. Развернуть проект можно в любом месте, на любой платформе. Все зависит только от универсальности вашего рецепта. Если он поддерживает один дистрибутив, вы можете сделать поддержку другого дистрибутива.

Бэкапы у нас тоже автоматизированы. Это отдельный ресурс, который так и называется — «задачи для бэкапа». Там мы указываем, что бэкапить, когда бэкапить, куда сохранять, сколько хранить и так далее.

Лего



- LVM
- Firewall
- Accounts
- Repos

hi⁺⁺

Что еще мы можем делать в рецепте. Мы можем управлять LVM-разделами. Например, для про-дакшн-окружения разделы будут большие, для девелоперских окружений разделы маленькие.

Мы можем открывать сетевые порты. Мы можем управлять аккаунтами, пользователями, подключать различные репозитории. Для системного администратора Chef-рецепт становится эдаким «Лего». Из этих блоков администратор может описать любой сервис.

Сервис

- создание разделов и директорий
- создание пользователей и групп
- поиск других хостов
- установка пакетов, создание конфигов и стартовых скриптов
- открытие портов, добавление проверок и задач для бэкапа
- настройка системы деплоя

hi⁺⁺

Вот пример простого сервиса. В итоге мы пришли к тому, что любой сервис можно описать следующим образом.

Создание разделов, создание дерева директорий.

Создание пользователей и групп.

Поиск других хостов.

Установка пакетов, создание конфигов, стартовых скриптов на основе всей предыдущей информации.

Открытие сетевых портов, добавление проверок и задач для бэкапа.

В конце — настройка системы деплоя, чтобы выкатить код непосредственно на сервер.

Chef как средство командной работы



- Система контроля версий
- Самодокументирование
- Общий стиль, подход, правила

hl++

Как можно использовать Chef как средство командной работы. Все рецепты, роли, окружения у нас находятся под управлением системы контроля версий. Новые возможности мы пишем в отдельных branch'ах, потом все это сливаем в master-ветку.

Документация, как настраивать сервер, не нужна. Chef рецепт + роль полностью описывает, как нужно настроить тот или иной сервис. Не нужны такие монструозные wiki, которые никогда не поддерживаются, и никто их не читает. Новый сотрудник может прийти прочитать рецепт и понять, как все происходит, как все настраивается.

В итоге, используя Chef в команде, вы вырабатываете общий стиль, общий подход. Вы можете делиться опытом с коллегами. Если коллега уходит в отпуск, вы можете продолжить его работу, потому что все сделано в одном стиле. Есть рецепты, роли, окружение. Все понятно, наглядно, все видно.

Knife. Абстрактное управление

- knife search node 'role:mysql' -r
- knife ssh "role:ruby" "sudo /etc/init.d/nginx-passenger status"
- knife ssh "role:ruby" interactive
- knife ssh "role:ruby" tmux
- knife ec2 server create -r "role[ruby]"
- knife exec -E 'nodes.all { |n| puts "#{n.name} has #(n.memory.total) free memory"}'



hl++

Александр Титов: Теперь я немного расскажу про абстрактное управление. Knife — это утилита Chef'a, которая может использовать его возможности по поиску по всему кластеру. Например, мы можем найти все ноды, у которых есть role:mysql и посмотреть на них. Это дает возможность абстрагироваться от конкретных нод, не держать в памяти, как называются ноды. Если это шардированный MySQL, там очень сложно удержать в памяти все знание о том, сколько шард на каком сервере.

Точно так же можно абстрактно запустить какую-то одну команду и на всех машинах с ruby back-end'ом узнать, запущен passenger или нет.

Можно сделать параллельный ssh на все машины с ruby (это следующая команда). Если ввести в конце interactive, можно параллельно на всех машинах выполнить какой-то набор команд. Например, выполнить passenger status и перезапустить его.

Также tmux (это такая фишечка). Я думаю, кто в теме — заценит. В tmux можно открыть сессии со всеми back-end'ами ruby.

Knife поддерживает плагины для создания серверов. В поставке уже идут плагины для amazon ec2, для redspace, еще нескольких облачных провайдеров.

Мы сейчас пытаемся добавить в Knife поддержку Cobbler'a. Мы делаем все совсем абстрактно. Можно сказать knife cobbler server create, сказать ему role, и сервер создастся. Для нас это следующий уровень абстракции.

Еще Knife позволяет выполнять отдельные куски кода на каких-то машинах. Например, мы можем посмотреть, сколько свободной памяти осталось по всем машинам. Это последняя команда. Сколько процессоров, вывести статистику или еще что-нибудь. По сути дела, это такое скриптование на весь кластер при помощи Ruby.

Полезные ссылки

- <http://wiki.opscode.com/display/chef/Home>
- <http://irimberman.nosterous.com/>
- <http://vagrantup.com/>
- <https://github.com/jedi4ever/veewee>
- <http://blog.frameos.org/>
- <http://blog.cyclecomputing.com/>
- <http://www.jedi.be/blog/>



Здесь немного полезных ссылок для тех, кто потом будет смотреть презентацию на Slide Share, когда ее выложат.

Вопросы?

КОНТАКТЫ:
mail: tiroff.a@gmail.com, igoritl@gmail.com
twitter: @osminog, @igoritl



Задавайте вопросы.

Вопросы и Ответы

Вопрос из зала: Как выглядит интерактив, если у вас 100 серверов с этой роли?

Александр Титов: Вывод будет очень плохо выглядеть. Весь std aut пойдет в одну консоль со всех машин. Будет много-много всего.

Вопрос из зала: Грубо говоря, если сказать knife ssh на все машины, он сказал cat/etc/passwd и потом сказал grep уже на моей машине, это реально?

Александр Титов: Нет, можно сказать grep на этой машине. Имеется в виду потом, после этого?

Вопрос из зала: Да.

Александр Титов: В целом, да. Но MapReduce он не умеет.

Вопрос из зала: MapReduce и не нужен в этом случае. Спасибо.

Вопрос из зала: Здравствуйте! Насколько я знаю, сейчас существует две независимые ветки Chef'a – 0.10 и 0.9. Отличия существенные? Стоит обновляться?

Александр Титов: Да, стоит. Собственно, Environments там нет, это очень полезная вещь для администрирования большой системы. Например, в версии 0.9.8 не работает нормально поиск. Там существенно переработаны выводы выполнения Chef Run, он становится более читаемым. Очень сильно переработан knife. Там сделана система плагинов. Системы плагинов нет в 0.9, надо много писать самому. Это из того, что я вспомнил навскидку.

Вопрос из зала: Совместимость какая-то существует?

Александр Титов: Да. Она полная. Естественно, клиент 0.9 может работать с сервером 0.10. Клиент 0.10 не может работать с сервером 0.9.

Вопрос из зала: Скажите, не страшно использовать на продуктиве такую систему? Там же нажал — и по всем ролям все просто остановилось.

Александр Титов: Да, страшно. (Смех, аплодисменты в зале). Было такое.

Вопрос из зала: Как контролировать?

Александр Титов: Контролировать можно на уровне Environments. Например, реальная проблема, которая была. Выкатывается Cookbook в продакшн, в Cookbook ошибка. Изменяется stage таким образом, что это кладет все сервера. Они высоконагруженные, много TCP-коннектов, Sys TTL это ограничивает. Было такое.

Единственный способ — тестирование перед этим. Я здесь не рассказывал то, что было в предыдущем докладе на РИТе. Считается, что это уже было. Для проверки рецептов мы используем Vagrant на своей машине. Мы это проверяем.

Использование Environments. Environments позволяют зафризить версию Cookbook на продакшне. Его невозможно залить именно с этой версии. Нельзя залить случайно. Это первое ограничение.

Второй плюс, который дает Environments. Можно создать gold next environments с несколькими машинами, на которые уже обкатывать выкатку новых Cookbooks и смотреть, что происходит. Потом это изменение накладывать на основной Environment. Это как раз то, что Environments дают для гибкости управления большой инфраструктурой.

Вопрос из зала: Понятно. В целом, можно сказать, что эта система стабильная?

Александр Титов: Система стабильная. Из-за самой системы не было ни одной проблемы. То, что было, это был человеческий фактор.

Вопрос из зала: Спасибо.

Вопрос из зала: Здравствуйте. У меня вопрос насчет костылей. Вы вначале упоминали. Чем помогает автоматизация от костылей, которые необходимо срочно применить. К примеру, какой-то деплой ломается в окружении из-за того, что нужно отключить какое-то шифрование. Это позволяет быстро применить рецепты на всех узлах?

Александр Титов: Да. Это раз.

Вопрос из зала: Как это связано с документированием? Вы не забыли, что нужно только здесь, на данном количестве нод, а в других не нужно.

Александр Титов: Можно применить на всех нодах — это раз. Во-вторых, часто костыли живут. Костыль один раз применяется, мы его закодили в нашем Cookbook'е. Дальше мы выкатываем новые сервера и забываем поставить туда этот новый костыль, например. Такая частая ситуация. Мы должны были это задокументировать. Документацию никто не ведет. Это все было в панике, потом была еще одна паника. Посередине забыли задокументировать этот костыль. Выкатили новые сервера — они не работают.

Ребята из Google рассказывали. Человек, который делал этот костыль, ушел в отпуск. Никто не помнит, как оно было сделано. На уровне Cookbook'ов это все кодируется. Во-первых, мы не забудем выкатить этот костыль на все сервера, чтобы все работало. Во-вторых, мы всегда помним, как мы его делаем.

Вопрос из зала: (Неразборчиво, без микрофона, 27:22).

Александр Титов: Мы пишем комментарии, и все. Еще к нему есть система markdown файликов. Read me mark что-то там. Там можно писать что-то типа wiki на каждый рецепт отдельно, как его использовать. Плюс комментарий внутри.

Игорь Курочкин: Вообще Chef-рецепты очень наглядные. Те ресурсы, которые вы используете (в языках программирования это аналог процедуры функций), очень наглядны. Directory — создать директорию, link — создать symlink, user — завести пользователя.

Ты смотришь на рецепт и видишь, что будет с сервером, как он будет настроен, какие директории он создаст. Только не нужно писать: «Создать 20 директорий и 20 блоков». Ты создаешь их в цикле — и все. Ты сразу видишь, как все создалось, какие пользователи создались, какие пакеты поставились, какие сервисы перезапустились. Сам рецепт очень наглядный. Это позволяет продолжать работу, совершенствовать его и так далее.

Вопрос из зала: Коллеги, спасибо! Очень познавательно. Навскидку на сотню серверов примерно оценка времени внедрения?

Александр Титов: В большинстве своем это все было сделано где-то за полгода.

Вопрос из зала: Вы этот путь прошли уже?

Александр Титов: Да, путь пройдет. Это все сейчас работает, это все Open Source. Не было сканено — для бэкапов мы используем Bacula. Все Open Source, все работает сейчас. Есть определенный уровень проблем, но он всегда сохраняется. Это намного лучше, чем когда этого нет.

Вопрос из зала: Полгода — это средняя оценка начать с нуля и прийти...

Александр Титов: Да. Три человека за полгода. Плюс еще помочь разработчиков.

Вопрос из зала: Я так понимаю, с нуля сервер невозможно развернуть. Есть чистая железка.

Александр Титов: Возможно.

Вопрос из зала: Каким образом? Должна быть привязка к хостеру, который предоставляет какие-то API для установки образа операционной системы...

Александр Титов: Нет-нет-нет. В целом вы немного не уловили. Я сейчас расскажу, как происходит. У нас есть железка. На железке вендором предустановлен RAID. Дальше Cobbler накатывает туда стандартный образ, делает стандартную разбивку LVM-разделов. Дальше накатывает туда Chef-клиент. Мы уже накатываем дополнительно роли, которые полностью настраивают сервер.

Сервер запускается, накатываются Cookbooks. Он добавляется в бэкап-систему, мониторинг-систему. На него устанавливается все, что нужно. Устанавливаются все проверки, связанные с сервисом. Дальше он поехал. Если нам надо переиспользовать этот сервер, мы его выключаем, заново накатываем все и используем его как другой сервис, например.

Вопрос из зала: Cobbler'ом накатываем систему. То есть мы в Cobbler прописываем Mac-адрес, все остальное заводится само?

Александр Титов: Именно. Mac-адрес, IP-адрес, дальше все само.

Игорь Курочкин: Причем для виртуальных машин это не нужно. Mac-адрес автоматически сгенерится, все поставится.

Вопрос из зала: Многие (37signals и другие) открывают свои рецепты. Ваши можно найти где-то, по Zabbix'у, в частности? Планируете, если нет открытых репозиториев?

Александр Титов: Вообще мы очень хотим, потому что мы сами не можем ловить баги, которые там есть. Просто ресурсов не хватает. Хотелось бы, чтобы кто-то другой поймал, а не мы.

Но мы сейчас дважды куплены, и с юридической точки зрения непонятно, можем мы выкладывать это в Open Source или нет. У этих рецептов нет достаточного уровня документации, чтобы их сообщество подхватило сразу же. Может быть, в ближайшее время мы их выложим, если нам разрешат. Мы очень хотим, на самом деле.

