1 Отчет о работе по тестированию и оценке времени доступа к сервису AccountService.

1.1 Введение. О сервисе.

Сервис Account Service позволяет хранить, получать и изменять данные о счетах пользователей посредством интернет соединения. Реализованы три программы:

- Account Service Server v1.0.

  Программа устанавливается на сервер. Хранит данные о счетах пользователей в базе данных под управлением СУБД MySQL. Связь сервера с СУБД осуществляется при помощи JDBC. Часть данных, к которым клиенты обращались в последнее время, кэшируется с помощью Google Guava Cache. Работа происходит в многопоточном режиме: один, главный, поток принимает входящие заявки от клиентов на известном и клиентам, и серверу порту, находит один из свободных обслуживающих потоков и сообщает клиенту порт, к которому необходимо подключиться для обслуживания. Обслуживающие потоки работают на порту, известном только стороне сервера. Возобновляя работу по команде главного потока, они принимают запрос от клиента, выполняют этот запрос и отправляют клиенту результат. При выполнении запроса на чтение данных сначала данные ищутся в кэше, а затем, если в кэше данных нет, выполняется запрос к базе данных, полученный результат помещается в кэш и отправляется клиенту. Также дополнительный поток выделен для ведения статистики. Поток просыпается только раз в 10 минут с целью записать статистику в файл.

- Account Service Client v1.0.

  Программа устанавливается на машину клиента. Позволяет передавать команды на сервер и получать ответ. Программа анализирует команду пользователя, затем производит все необходимые запросы на сервер и выводит ответ без участия пользователя.

- Account Service Test Client v1.0.

  Программа устанавливается на машину клиента. Установка программы не является необходимым условием корректной работы сервиса. Служит в первую очередь для тестирования сервера при большой нагрузке. Позволяет в многопоточном режиме практически одновременно отправлять несколько запросов на сервер.

1.2 Тестирование.

Тестирование можно разделить на две части: тестирование работоспособности и тестирование времени работы. Для тестирования работоспособности использовался графический интерфейс phpMyAdmin, позволяющий

следить за изменениями в базе данных, а также Account Service Client v1.0 и Account Service Test Client v1.0.

Практическое тестирование времени работы на локальной машине в случае больших нагрузок не представляется возможным, поскольку клиентские процессы также имитируются локальной машиной, что создает существенное увеличение времени работы как за счет клиентских процессов, так и за счет переключения между ними и серверными процессами.

Тестирование работоспособности.

Работоспособность одного клиента была протестирована и не вызвала сомнений. Значительно больший интерес представляет тестирование сервиса под большой нагрузкой с использованием Account Service Test Client v1.0.

Синтаксис команд Account Service Test Client v1.0:

<type> <minId> <maxId> <repeat> <value>

<type> — тип команды, может быть add или get.

<minId> — минимальный идентификатор, к которому будет применена команда.

<maxId> — максимальный идентификатор, к которому будет применена команда.

<repeat> — количество раз, которое каждая команда будет повторена.

<value> — число, которое будет добавлено командами add, если тип команды add.

Команда выполнит <repeat> раз запрос <type> <id> <value> для всех <id> между <minId> и <maxId>.

Все проведенные тесты можно разделить на 3 типа:

- Тест корректности многопоточного добавления. Тестовая команда:

  add x x N v, где N — некоторое большое число, x и v — произвольные числа.

- Тест стабильности работы команды get при большой нагрузке. Тестовая команда:

  get 1 N 1, где N — некоторое большое число.

- Тест стабильности работы команды add при большой нагрузке. Тестовая команда:

  add 1 N 1 v, где N — некоторое большое число, v — произвольное число.

Результаты тестов:

- Корректность многопоточного добавления. При $N = 500, 1000, 2000$ программа работает корректно, при $N = 4000$ программа работает корректно, но уже порядка 10 минут. Дальнейшие тесты целесообразны только при наличии различных серверной и тестовой машин.

- Корректность работы команды get при большой нагрузке. При $N = 4000$ программа работает корректно, уже начиная с $N = 6000$ программа работает крайне медленно.
- Корректность работы команды add при большой нагрузке. Показатели корректной работы схожи с первым случаем.

1.3 Оценка времени работы.

Как уже отмечалось ранее, тестовый клиент в рамках локальной машины не способен адекватно отразить время работы сервиса. Поэтому все практические данные, которые можно извлечь из тестирования, сводятся к времени обработки одного запроса.

Время обработки add-запроса.

Размер базы данных, на которой проводилось тестирование, — порядка 20000 записей. С помощью измерения времени работы разных участков программы было выяснено, что практически все время работы программы уходит непосредственно на исполнение SQL-запроса (в разных ситуациях от 500 до 4000 мс) , и лишь малая часть (менее 100 мс) на передачу данных, анализ запроса и прочее. С помощью графического интерфейса phpMyAdmin было дополнительно проверено, что запросы к базе данных выполняются действительно столько. Таким образом, узким местом программы является единственный SQL update-запрос, оптимизация которого находится в руках разработчиков СУБД. Далее мы будем предполагать, что среднее время обработки add-запроса примерно равняется 2000 мс.

Время обработки get-запроса.

Основное принципиальное отличие get- от add- запросов заключается в кэшировании, поэтому подавляющее большинство get-запросов обрабатываются на порядок быстрее add-запросов. С учетом знаний о данных, хранимых в кэше, была предпринята попытка отправить get-запрос по идентификатору, не хранимому в кэше. Однако и такой запрос был выполнен практически мгновенно (около 50 мс), что косвенно подтверждает выдвигаемые различными источниками предположения о внутреннем кэшировании СУБД. Разумно предполагать, что большинство get-запросов будет попадать в кэшированные данные, поэтому будем считать, что среднее время обработки get-запросов примерно равняется 50 мс.

Работа нагруженного сервера.

Алгоритм многопоточной обработки клиентов.

- Во время запуска сервера инициализируется N обслуживающих потоков. Они организуются в очередь свободных потоков и запускаются. Каждый поток работает бесконечным циклом:
  - Ждет, пока поступит команда возобновить работу.
  - Обслуживает поступившего клиента.
  - Сообщает основному потоку, что закончил обслуживание и освободился.

- Основной поток постоянно прослушивает известный клиенту порт. Как только подключается клиент, сервер берет из стека свободный поток, сообщает клиенту порт, на котором работает этот поток, и возвращается к прослушке новых клиентов. В случае занятости всех портов сервер просит клиента подождать. Клиент ждет случайное число от 1 до 50 мс.

Такая схема обслуживания позволяет:

- При правильной конфигурации полностью задействовать ресурсы сервера и не создавать потоков больше, чем может исполняться параллельно.
- Минимизировать вычисления во время работы сервера: все предварительные данные собираются на этапе запуска, новые потоки не создаются и не запускаются.
- Переложить максимально большую часть работы сервера на N обслуживающих потоков с одного слушающего.

Оценка времени доступа сервиса.
Мы уже предположили некоторые параметры работы сервера:

- add-запросы выполняются в среднем $t_{add} = 2000$ мс.
- get-запросы выполняются в среднем $t_{get} = 50$ мс.

Дополнительно предположим:

- Машина сервера имеет возможность запускать параллельно $N+1$ процесс, то есть N обслуживающих потоков исполняются параллельно. Потоком статистики, просыпающимся раз в 10 минут, пренебрегаем. Будем считать, что $N = 20$.

Уже исходя из этих предположений можно сделать некоторые выводы.

- В секунду в среднем обрабатывается $\frac{N \cdot 1000}{t_{add}} = 10$ add-запросов.
- В секунду в среднем обрабатывается $\frac{N \cdot 1000}{t_{get}} = 400$ get-запросов.

Однако, эта информация не является полезной для клиента: он хочет знать *свое* время ожидания.

Для вычисления подобной информации необходимы предположения о порядке поступления запросов от клиентов. Жизненные ситуации можно приближенно разделить на два типа:

- Регулярные ситуации. Новые запросы поступают равномерно, можно сделать некоторые предположения регулярности о потоке запросов.
- Критические ситуации. По некоторым причинам в короткий промежуток времени поступает большое количество запросов.

Регулярные ситуации.

Мы рассмотрим лишь случай поступления заявок одного типа. Подобные случаи достаточно точно описываются классической моделью системы массового обслуживания с бесконечной очередью. Предполагается, что время до появления нового запроса распределено экспоненциально с известным матожиданием, время обработки одного запроса одним потоком также распределено экспоненциально с известным матожиданием. В случае, когда новый запрос попадает не в очередь, а сразу обрабатывается, среднее время ожидания будет равняться t_{add} и t_{get} соответственно. Столь же очевидно, что если на сервер будет *регулярно* поступать заявок больше, чем он может обработать, очередь будет разрастаться до бесконечности. Ключевым параметром в данной системе является $q = \frac{t_{income}}{Nt_{serve}}$, где t_{income} — среднее время появления нового запроса, t_{serve} — среднее время обработки запроса одним потоком. Вероятность оказаться i-ым в очереди на обработку будет убывать как геометрическая прогрессия со знаменателем q. При q близких к единице, хотя и меньших ее, будет наблюдаться эффект длинной очереди из-за неравномерности поступления запросов: некоторую часть времени заняты будут не все потоки, а некоторую часть — поступает несколько больше, чем обычно, запросов, поэтому накапливается очередь. Впрочем, поскольку точно предсказать q невозможно, а $q \geq 1$ приведет к полной неработоспособности сервера, в реальной жизни необходимо потребовать не просто $q < 1$, а, скажем, $q < 0.8$. Также стоит вспомнить, что попадание в очередь даже на N-ое место приведет к ожиданию только $2 \cdot t_{serve}$ вместо t_{serve} в случае не загруженной системы. Поэтому при достаточно больших N и отделенным от единицы $\frac{t_{income}}{Nt_{serve}}$ среднее время ожидания не будет существенно отличаться от t_{add} и t_{get} соответственно.

Критические ситуации.

Мы будем рассматривать упрощенную модель, когда одновременно на сервер поступает M запросов одного типа, а затем запросы не поступают. В идеализированном случае процесс представляется очевидным: раз в t_{serve}, где t_{serve} обозначает время обработки запроса данного типа, N очередных запросов обслуживаются, начинают обслуживаться следующие N. В таком случае первая группа из N запросов будет обслужена за t_{serve}, вторая — за $2t_{serve}$, ..., $\frac{M}{N}$-ая — за $\frac{M}{N} \cdot t_{serve}$. То есть среднее время ожидания будет $\frac{M}{2N} \cdot t_{serve}$, максимальное — $\frac{M}{N} \cdot t_{serve}$. Если взять наши данные времени обработки запроса и потребовать, чтобы среднее время ожидания не превышало 5 секунд, то получим, что успешно обрабатываются одновременно поступающие 100 запросов add или 4000 запросов get.

Однако, идеальная ситуация далека от реальной. Во-первых, в идеальной ситуации сервер сам должен вызывать ближайшего абонента из очереди, во-вторых, одновременное обращение большого числа клиентов затрудняет работу основного потока сервера. Обе проблемы решаются ожиданием клиента в случае занятости все обрабатывающих потоков. Однако, актуальным остается вопрос: сколько ждать? Этот вопрос решен в пользу случайности. Клиент ждет случайное число миллисекунд от 1 до 50. Преимуществ

у такого подхода два. Во-первых, заранее предсказать время. за которое освободится очередной обрабатывающий поток, непросто, а для большей производительности нужно, чтобы как можно скорее после освобождения обрабатывающего потока поступил очередной запрос. Во-вторых, в случае одновременного обращения большого числа клиентов к одному слушающему потоку могут возникнуть дополнительные сложности и задержки в обработке всех запросов. Случайное время ожидания позволяет сделать заявки равномерно распределенными во времени. Таким образом реальные показатели можно считать максимально приближенными к идеальным.

Итак, в условиях допущений работоспособности машины и пренебрегая скоростью соединения (соединение с localhost практически мгновенное) можно получить следующие оценочные параметры:

- Среднее время обработки add-запроса: 2000 мс.
- Среднее время обработки get-запроса: 50 мс.
- Среднее количество обрабатываемых add-запросов при максимальной загруженности: 10 в секунду.
- Среднее количество обрабатываемых get-запросов при максимальной загруженности: 400 в секунду.
- Рекомендуемая максимальная нагрузка на сервер: 8 запросов add или 320 запросов get в секунду.
- Среднее время ожидания до выполнения запроса при соблюдении рекомендуемой максимальной нагрузке: не многим более 2000 мс и 50 мс для add- и get- запросов соответственно.
- Максимальное количество одновременно поступивших запросов при требовании среднего времени ожидания выполнения не более 5 секунд: 100 add- и 4000 get- запросов соответственно.

Показатели, влияющие на быстродействие сервера:

- Тактовая частота одного вычислительного потока сервера. Влияет на быстродействие линейно.
- Размер базы данных. Точных сведений о том, насколько существенно уменьшается скорость выполнения запросов при увеличении числа записей, нет, но есть основания предполагать, что не более чем логарифмически от количества записей в базе данных.
- Скорость соединения. В разумных пределах скорость соединения не окажет существенного влияния на add-запросы, однако, может существенно повлиять на скорость выполнения get-запросов вплоть до того, что может стать узким местом таких запросов.

В зависимости от прикладных условий приложения могут быть разработаны модификации. Некоторые из прикладных условий, при уточнении которых может быть разработана эффективная модификация сервиса:

- Соотношение количества поступающих add- и get- запросов.
- Критичность срочного выполнения add-запросов.
- Предполагаемая повторяемость add- запросов по одному идентификатору.

Кирилл Савенков,
ostrich@flyingsteps.org
02.08.2014