Министерство образования и науки Российской Федерации

МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ (государственный университет)
ФАКУЛЬТЕТ УПРАВЛЕНИЯ И ПРИКЛАДНОЙ МАТЕМАТИКИ
КАФЕДРА ТЕОРЕТИЧЕСКОЙ И ПРИКЛАДНОЙ ИНФОРМАТИКИ

Направление подготовки: 03.03.01 Прикладные математика и физика

“РАЗРАБОТКА ПОДСИСТЕМЫ МОНИТОРИНГА ПРОИЗВОДИТЕЛЬНОСТИ ПРИЛОЖЕНИЙ В AMAZON WEB SERVICES ”

Выпускная квалификационная работа
(бакалаврская работа)

Выполнил:
студент 376 группы _____Жадченко А.В.
Научный руководитель _____Дяйкин А.Д.
Научный консультант _____Кулага А.А.

Москва, 2017

1. Введение **3**

2. Постановка проблемы **6**

2.1 Необходимые понятия 6

2.1.1 Инструментирование 6

2.1.2 Транзакция 7

2.1.3 HTTP и REST API 7

2.2 Формулирование проблемы 8

2.3 Задача 8

3. Описание модели **9**

4. Решение **10**

4.1 Тестовый сервер 10

4.1.1 Внешнее представление 10

4.2 Инструментация 12

4.2.1 Инструментирование flask 12

4.2.2 Инструментация HTML 13

4.3 Сервер сбора и обработки данных 14

4.3.1 Получение данных 14

4.3.2 База данных для хранения информации 14

4.3.3 Сборка транзакций. Проблемы 16

4.3.4 Нетривиальные транзакции 20

5. Результаты работы **21**

5.1 Примеры полученных данных 22

Список литературы **23**

Приложения **24**

Приложение 1. OPTIONS, GET, POST 25

Приложение 2. Обработка POST, GET запросов 27

Приложение 3. Инструментация flask 28

Приложение 4. Инструментация вызовов базы данных 29

Приложение 5. Инструментация Javascript 30

Приложение 6. Поддержка cross site scripting 31

Приложение 7. Нормализация транзакции из 4 временных отметок 32

Приложение 8. Генерирование графика транзакции 34

Приложение 9. Генерирование графика загрузки страницы 35

1. Введение

В настоящее время практически невозможно представить себе успешную продаваемую услугу или товар, не имеющих сайта или иного размещения в сети Интернет. Это не только удобно для покупателей, но и позволяет серьезно снижать расходы, что приводит как к появлению новых сервисов, ориентированных на интернет, так и к эволюции старых моделей, например появлению интернет-магазинов. У интернет-магазинов значительная часть бизнес-процесса перенесена в цифровую область - они не арендуют помещения для организации продажи, экономят на найме кассиров и консультантов, имеют, как правило, только склад и(или) точку выдачи.

В виртуальной битве за клиентов опыт взаимодействия пользователя с сайтом компании имеет серьезный вес. Клиенты не только ожидают приемлемый уровень сервиса, но и в условиях высокой плотности рынка готовы переходить между конкурентами в поисках различных вариантов решения своей проблемы. В связи с этим возникает необходимость не только иметь сайт, но и поддерживать его и, как следствие, всю инфраструктуру на приемлемом уровне, иначе это может обернуться потерей и недополучением прибыли. Кроме того, остро строит проблема оценивания долей положительного и отрицательного опыта пользователя, их соотношение на протяжении значительных промежутков времени.

Уже существуют некоторые программные решения для решения этих задач, кроме того, рынок этих продуктов - Application Performance Monitoring, активно развивается и по оценкам экспертов является одним из самых быстрорастущих в мире - около 15% в год. Ожидается, что объем рынка, в 2016 году равный 3.1 миллиардов долларов США, к 2020 превысит 5 миллиардов долларов (Рис 1. [1]).

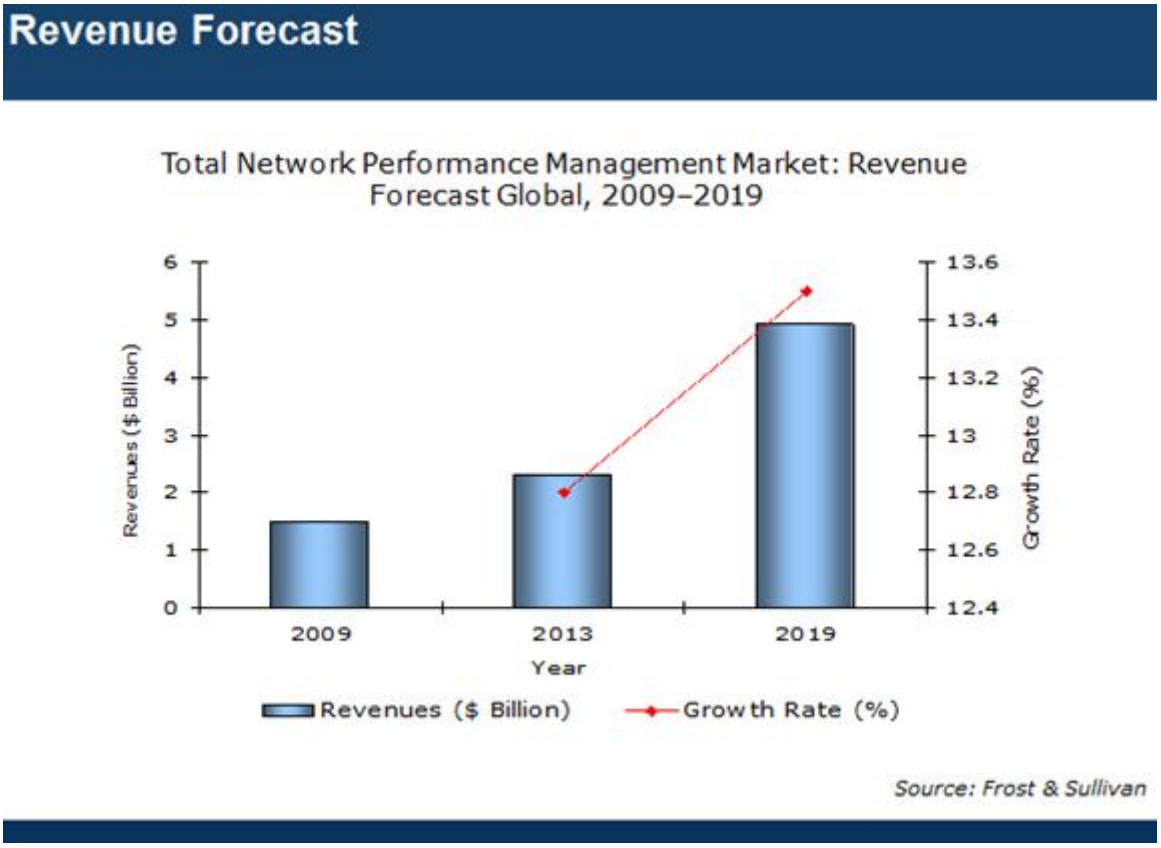

Рис 1. Прогнозирование роста рынка APM

В данный момент принято разделять сферу Application Performance Monitoring на пять категорий (см. Рис. 2):

1. Опыт пользователя - мониторинг метрик, видимых и влияющих на конечного пользователя, например время загрузки страницы.
2. Мониторинг архитектуры - определение производительности отдельных компонентов информационной системы
3. Бизнес транзакции - влияние производительности отдельных частей и их вклад в общее время запроса, производимого пользователем
4. Глубокая диагностика - определение проблем внутри недостаточно производительных модулей, вплоть до определенных участков кода, слежение за ошибками, происходящими на самом низком уровне
5. Отчеты и аналитика - представление собранной информации в удобном и понятном виде. Определение трендов, устойчивости информационной системы, определение критических значений.

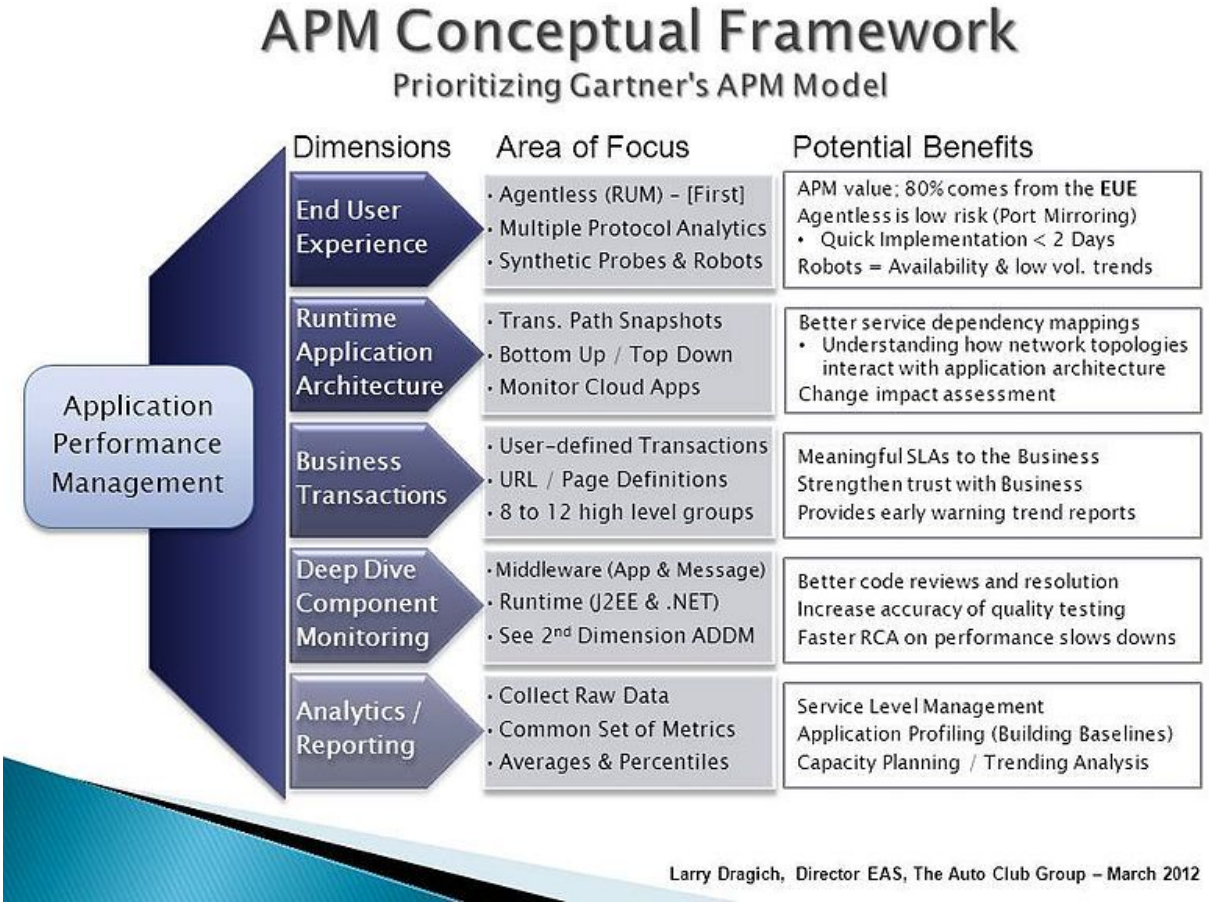

Рис 2. Application Performance Management Categories

2. Постановка проблемы

2.1 Необходимые понятия

2.1.1 Инструментирование

В области программирования под инструментированием понимают возможность отслеживания или установления количественных параметров уровня производительности программного продукта, а также возможность диагностировать ошибки и записывать информацию для отслеживания причин их возникновения.

Измерение в виде инструкций кода обычно используется для отслеживания работы определенного компонента системы (например, инструкции, выводящие логи на экран). Когда приложение содержит инструментальный код, им можно управлять при помощи специальных инструментов-утилит. Измерение необходимо для оценки производительности приложения. Методы измерений делятся на два основных типа: измерения на основе исходного кода и измерения на основе двоичного кода.

В области программирования измерение означает возможность измерить приложение с точки зрения следующих параметров:

- Трассировка кода — получение информационных сообщений о выполнении приложения на всем протяжении его работы.
- Отладка программы и (структурированная) обработка исключений — отслеживание и исправление ошибок программистов в приложении ещё на стадии его разработки.
- Профилирование — набор методик отслеживания производительности кода, включая измерение.
- Счётчики производительности — компоненты, позволяющие отслеживать уровень производительности приложения.

- Регистраторы событий — компоненты, позволяющие получать уведомления и отслеживать ключевые события при выполнении приложения.

2.1.2 Транзакция

Под транзакцией понимается неделимое с точки зрения пользователя действие - например ввод логина, пароля и нажатие кнопки отправить. Хотя с точки зрения информационной системы как минимум помимо обращения к веб-серверу для обработки запроса и придется сделать обращение к базе данных.

2.1.3 HTTP и REST API

REST (сокр. от англ. Representational State Transfer — «передача состояния представления») — архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы. В определённых случаях (интернет-магазины, поисковые системы, прочие системы, основанные на данных) это приводит к повышению производительности и упрощению архитектуры. В широком смысле компоненты в REST взаимодействуют наподобие взаимодействия клиентов и серверов во Всемирной паутине.

HTTP (англ. HyperText Transfer Protocol — «протокол передачи гипертекста») — протокол прикладного уровня передачи данных (изначально — в виде гипертекстовых документов в формате «HTML», в настоящий момент используется для передачи произвольных данных) [2]. В Приложении 1 рассмотрены используемые запросы (GET, POST, OPTIONS).

2.2 Формулирование проблемы

В связи с критической зависимостью практически любого бизнеса от IT инфраструктуры необходимо не только четко осознавать внутреннее состояние сложной информационной системы в любой момент времени, но и иметь всесторонние данные об этой системе с точки зрения конечного пользователя.

2.3 Задача

Задачей является построение прототипа системы Application Performance Monitoring и тестирование его на соответствие экспериментальной модели. С помощью прототипа должно быть возможным сбор и анализ данных сложных информационных систем, построение метрик, соответствующих опыту конечного пользователя.

3. Описание модели

Логически модель разделена на три части:

1. Инструментирование

Фрагменты кода, относящиеся к специфическому языку или среде разработки, позволяющие собирать различные данные при вставке их в соответствующие модули.

В работе используется инструментация для языков программирования Python, Javascript, фреймворка Flask;

2. Серверный модуль для сбора метрик

Предоставляет API для приема HTTP метода POST с заданными параметрам. Поддерживает междоменные запросы. Пишет в базу данных все полученные части и заголовки транзакций;

3. Модуль для обработки и аналитики

Содержит функции для работы с заголовками и частями транзакций. Умеет правильно компоновать данные и отличать валидные транзакции от невалидных.

Содержит алгоритмы для нормировки транзакций, подсчета их суммарного времени и простоя.

Кроме того, умеет генерировать данные для постройки графиков метрик и транзакций.

4. Решение

4.1 Тестовый сервер

В качестве инструментируемой информационной системы выступал написанный в ходе выполнения квалификационной работы веб-сервер, состоящий из конфигурации nginx + UWSGI + python::flask + sqlite3 + AWS API к Amazon S3 и являющийся модельным представлением реальной системы, состоящей из веб-сервера, базы данных и удаленного объектного хранилища.

4.1.1 Внешнее представление

Для пользователя через браузер доступны следующие четыре веб страницы:

1. / - главная страница, состоящая из приветствия и навигации. Поддерживаются GET запросы.
2. /workload - страница для запуска нагрузки (циклический подсчет числа пи) на сервере течение заданного времени для эмуляции нехватки вычислительных ресурсов. Поддерживаются GET, POST запросы
3. /dbrequest - страница, предоставляющая интерфейс для базы данных сервера, в котором можно получить значение по ключу. Поддерживаются GET, POST запросы
4. /bucketrequest - страница, предоставляющая интерфейс для Amazon Simple Storage Service, с помощью чего можно получить файл, хранимый на AWS S3. Поддерживаются GET, POST запросы

4.2.2 Архитектура сервера

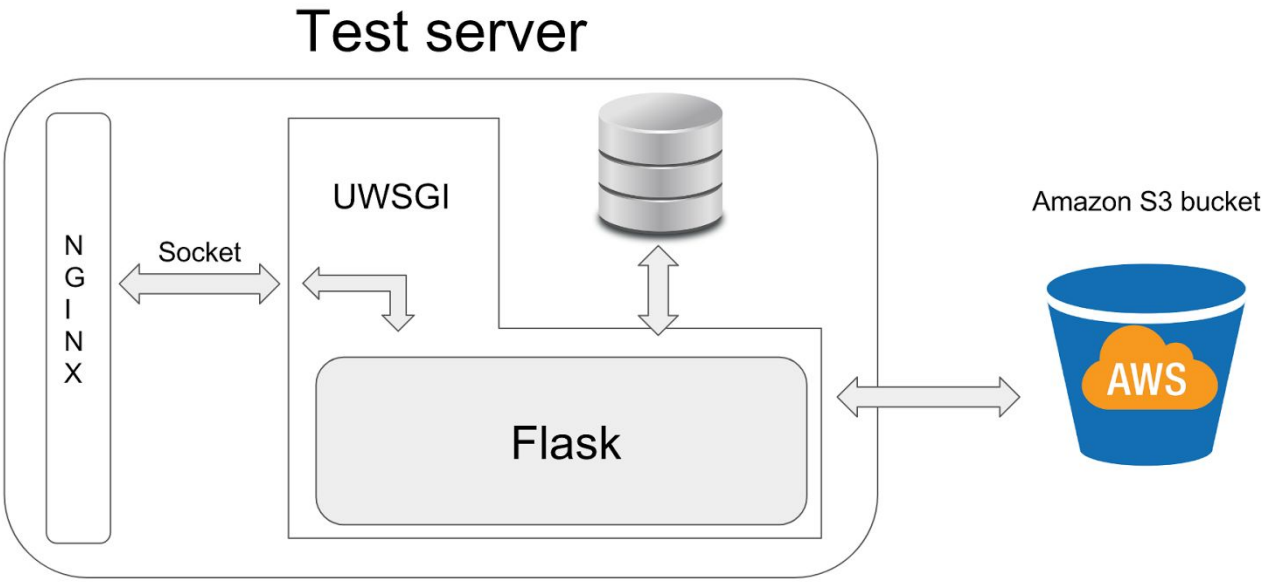

Рис. 2 Архитектура сервера

Nginx отвечает за прослушивание портов, соединение с клиентами и передачу данных.

UWSGI представляет собой прослойку, которая может запускать Python, в том числе динамически увеличивая или уменьшая количество потоков, которые одновременно могут обрабатывать запросы, в зависимости от нагрузки. Также UWSGI перенаправляет данные, полученные от nginx в flask и обратно.

Flask - фреймворк [3,4], с помощью которого непосредственно написана обработка запросов пользователя (см. Приложение 2) и генерация страниц.

В качестве базы данных используется sqlite3.

Также непосредственно в приложении при соответствующем запросе устанавливается соединение с AWS S3 для поиска и скачивания файла.

4.2 Инструментация

4.2.1 Инструментирование flask

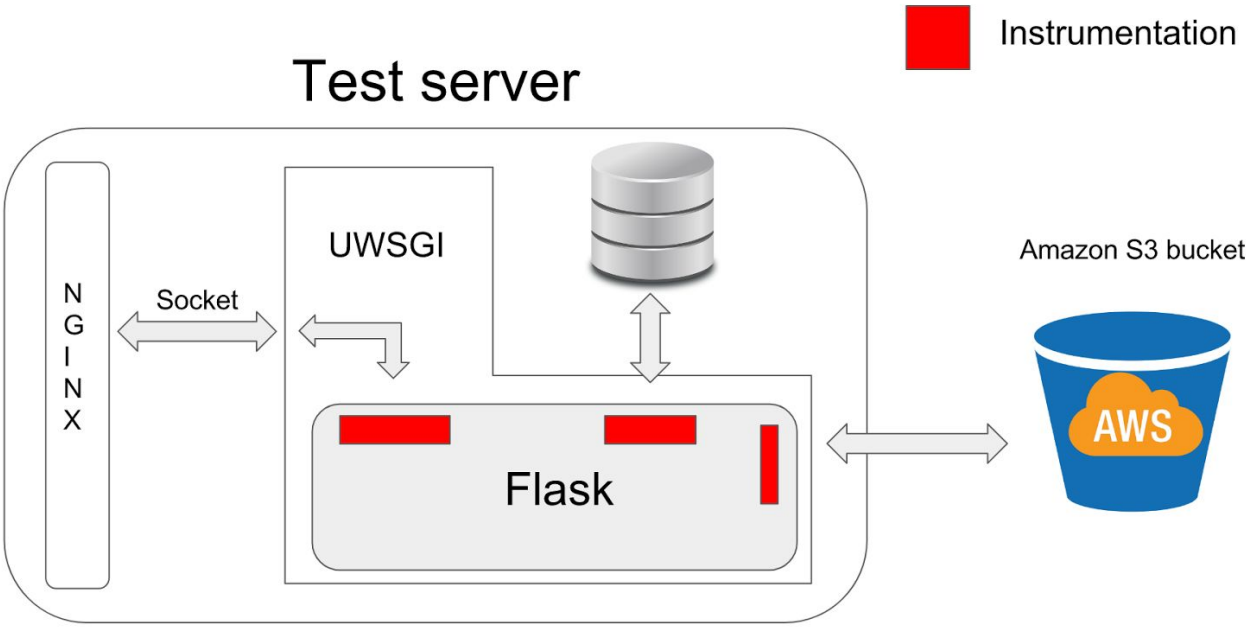

Рис. 3 Архитектура со схематическим изображением инструментированных частей

Перехват проходящих данных производится с помощью реализации декораторов @app.before_request и @app.after_request, что можно увидеть в Приложении 3. Такой подход дает доступ ко всем данным как и запроса, так и ответа, также позволяет их модифицировать до того, как они дойдут до адресата.

Получение данных о производительности sqlite достигается путем добавления кода, обрамляющего запросы в БД (Приложение 4)

Получение данных о производительности AWS S3 производится аналогично случаю с базой данных.

Собранные данные помечаются уникальными идентификатором - UUID и посылаются на обработку.

4.2.2 Инструментация HTML

Для получения данных с компьютера пользователя инструментируется каждая html страница, отдаваемая сервером. Браузер при загрузке страницы исполняет заранее подготовленный скрипт, который позволяет получать различную информацию о компьютере пользователя, соединении и производительности.

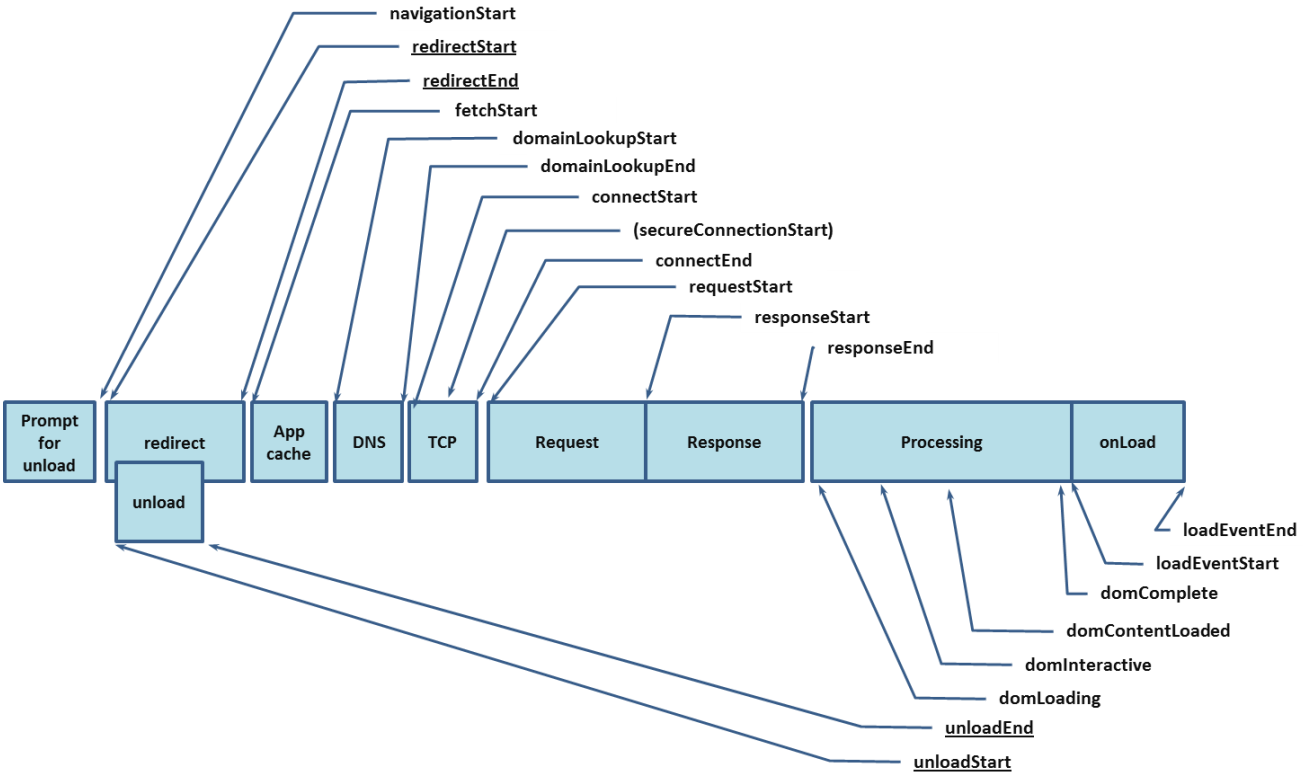

Рис. 4 Стандарт Navigation Timing

В квалификационной работе используется стандарт JavaScript Navigation Timing [5]. Согласно ему, браузер заполняет специальные переменные числами, равными Unix-времени - количеству миллисекунд, прошедших с 1 января 1970 года, на момент начала соответствующего события. Собранные данные помечаются уникальным идентификатором UUID и отсылаются на сервер для обработки. Скрипт, используемый в работе, можно найти в Приложении 5.

4.3 Сервер сбора и обработки данных

4.3.1 Получение данных

Отправка и получение данных производится с помощью метода POST реализуемого HTTP. Однако в связи с защитой от атак Cross Site Scripting, в настоящее время на запросы, посланные с открытой в веб-браузере страницы не на сервер, с которого она была отдана, налагаются ограничения. А именно, принимающая сторона должна на запрос OPTIONS (см. Приложение 6), который посылается до кросс-серверного POST, ответить со специальными заголовками, в которых указывается допустимые типы запроса, диапазон адресов, откуда могут посылаться запросы, и допустимые страницы назначения. Только тогда, браузер, проверив соответствие присланных заголовков, отсылает POST запрос.

В работе запросы могут прийти в одном из двух видов - заголовок или часть транзакции.

В заголовок входит различная метаинформация, присущая запросу - IP пользователя, время начала, браузер и так далее.

В части транзакций входит UUID транзакции, а также одна или более записей вида *время:действие*.

4.3.2 База данных для хранения информации

Хранение и обработка собранных данных организовано в локальной базе данных sqlite[6]. База данных имеет две таблицы соответственно для частей транзакций и заголовков (Рис. 5).

Все получаемые данные напрямую записываются в базу данных. Обработка происходит на поздних этапах для проведения анализа и создания графиков.

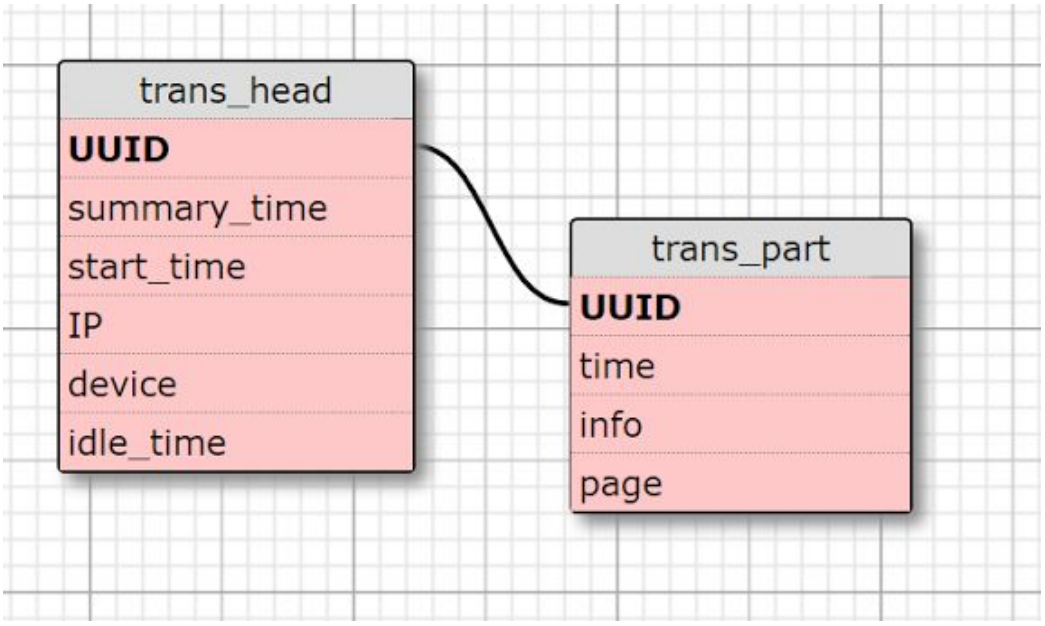

Рис. 5 Схема базы данных

Таблица trans_head представляет собой заголовки транзакций и состоит из следующих полей:

1. UUID[7] - уникальный идентификатор транзакции, использующийся для поиска и обработки данных
2. summary_time - суммарное время, в течение которого транзакция работала. Не учитывается idle_time. Проставляется в процессе компоновки транзакции.
3. start_time - время начала транзакции. Может поменяться в процессе компоновки транзакции.
4. IP - адрес компьютера, которого пришел запрос. Может использоваться для геолокации.
5. device - заголовок user-agent из http запроса. С помощью него можно идентифицировать, как правило, разрядность системы, платформу и браузер.
6. idle_time - время, в течение которого информационная система ожидала действия пользователя. Не учитывается в суммарном времени транзакции. Проставляется в процессе компоновки.

Таблица trans_part представляет собой части транзакций и состоит из следующих полей:

1. UUID - уникальный идентификатор транзакции, использующийся для поиска и обработки данных.
2. time - временная метка, которая представляет собой число секунд, прошедших с 1 января 1970 года. Однако стоит учитывать, что в частях транзакции, пришедших с браузера, время считается в миллисекундах с 1 января 1970 года. Во время нормализации время меняется на секунды и, возможно, немного сдвигается.
3. info - информация о временной метке. Содержит строку строго заданного формата и определяет компоненту, откуда пришла часть транзакции.
4. page - страница, на которую был послан запрос.

4.3.3 Сборка транзакций. Проблемы

При наличии правильного корректного UUID группировка частей транзакций в одну не представляет проблемы. Но при этом следует учитывать, какие транзакции все же являются корректными, а какие - нет. Также остро стоит задача корректного подсчета времени транзакции.

Первая проблема - нехватка симметрично расположенных частей. Например, в базе у одного UUID есть часть транзакции, которая помечена как начало (его отсылка на сервер) GET запроса, а часть транзакции, соответствующая его окончанию (ответ) отсутствует. В таком случае транзакция признается некорректной и удаляется из базы данных.

Однако, в данном случае следует отличать нехватку симметричной части от отсутствия вышележащей или вложенной части. Вполне понятно,

что результаты временных меток сервера по обработке запроса будут содержаться в отрезке из временных меток, пришедших от браузера.

Рассмотрим простейший запрос GET на сервер на /. В его результате пользователь получит страницу. В базу данных у нас попадут четыре метки - две со стороны сервера о начале и окончании обработки запроса и две со стороны пользователя о посылке запроса и приеме ответа.

Действительно, сначала запрос формируется у пользователя - 1 метка Request_start, затем метка GET_start, затем GET_end и наконец Request_end. В стандарте Navigation Timing она обозначается как Response_end, однако, для упрощения мы далее будем ее называть Request_end.

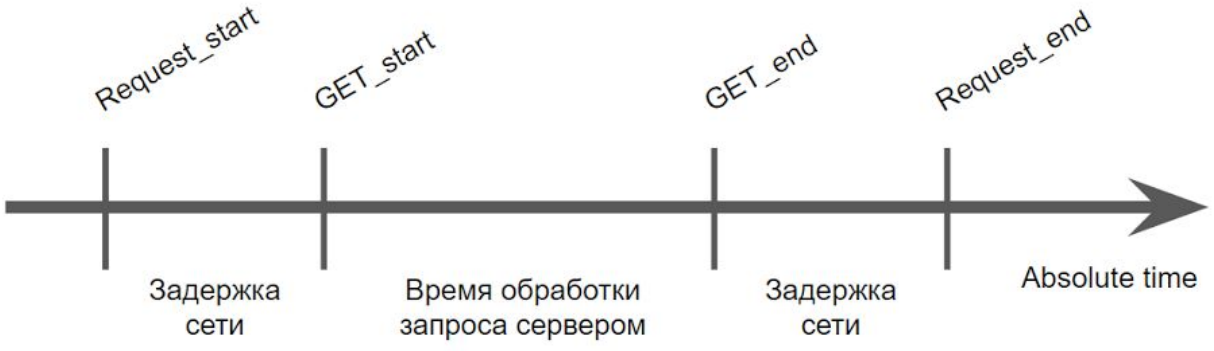

Рис. 6 Временные метки простого запроса

При этом, если у пользователя в браузере отключен JavaScript, то мы не получим метки Request_start и Request_end. Такая возможность существует и такая транзакция будет являться корректной, хоть и не полной по данным. Ее также следует аккуратно учитывать при анализе различных метрик, как, например, время загрузки страницы у пользователя и время обработки запроса сервером.

Вторая важная проблема заключается в том, что время на компьютере пользователя и сервера не синхронизировано. В оптимистичном варианте у пользователя просто стоит другой часовой пояс, и его можно попробовать угадать по геолокации или IP адресу.

Но на самом деле все намного хуже - время на одном из компьютеров может идти вперед на пару секунд. На Рис 6. было указано абсолютное время, однако реально в локальном времени сервера по сбору метрик транзакция без обработки времени может выглядеть как на Рис 7.

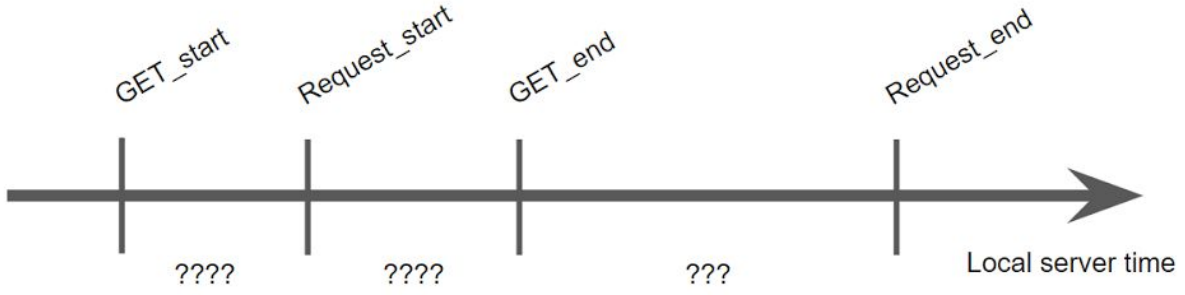

Рис. 7 Временные метки на локальном компьютере, нарушение причинно-следственной связи

К сожалению, такая ситуация довольно распространена. Поэтому каждую приходящую транзакцию при обработке нормировать и восстанавливать причинно-следственную связь. Для этого мы воспользуемся тем, что отрезок времени, прошедший между отправкой запроса и получением ответа больше, чем время обработки запроса внутри сервера, и вложим один в другой. К сожалению, в этом варианте мы получим усреднение по задержке сети между метками (Рис 6.), но этим можно пренебречь, так как оно в общем случае не изменится за доли секунды, пока сервер обрабатывает запрос или же несравненно мало по сравнению с этим временем. Нормализация показана в Приложении 7.

В наихудшем варианте может случиться так, что на одном из компьютеров время течет быстрей относительно другого - неточный кварцевый резонатор, по которому идут часы, ошибка с синхронизацией времени на виртуальной машине, разгон материнской платы до скоростей, сравнимых со световой и т.д. В данном случае временные метки будут выглядеть, например, как на Рис. 8.

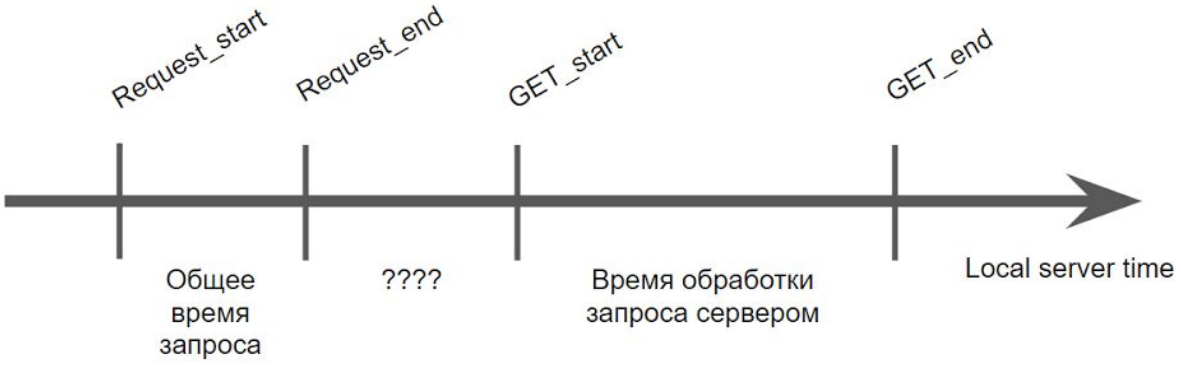

Рис. 8 Невосстановимое нарушение причинно-следственной связи

Характерное отличие таких записей в том, что не соблюдается причинно-следственная связь, т.е. вложенная, основываясь на логике работы, часть оказывается по факту больше той, в которую вложили. Например, время обработки на сервере больше общего времени запроса.

Такие транзакции необходимо помечать и не учитывать в выборках для составления метрик.

Также в эту категорию попадают различные другие ошибки с невосстановимым нарушением причинно-следственной связи, как, например, идущее назад время, в результате которого завершающая метка окажется впереди начинающей.

4.3.4 Нетривиальные транзакции

В категорию нетривиальных транзакций мы будем включать все транзакции, в течение которых напрямую был задействован пользователь. В том числе получение уникально генерируемых страниц, на которых необходимо ввести некие данные и отправить их обратно, например, подтверждение банковского платежа по двухфакторной аутентификации. В данном случае необходимо выделить время, которое пользователь тратит на ввод данных. Его нужно не учитывать при создании метрик, отражающих скорость работы информационной системы.

При этом, конечно, с точки зрения пользовательского опыта эта величина является важной и тоже подлежит внимательному изучению. Кроме того, важным является, смог ли пользователь довести до конца эту транзакцию или вышел с сайта.

Реализация этого на тестовом сервере - страница с запросом в базу данных. В информации в одну транзакцию объединены GET и POST запросы.

5. Результаты работы

- Проанализирован рынок систем класса Application Performance Monitoring
- Решены архитектурные и технические проблемы, связанные с реализацией программного комплекса APM
- Найдены способы инструментации для популярных сред разработки и языков программирования
- Реализован полный прототип системы APM
- Полученные в ходе эксплуатации данные соответствуют теоретической модели
- Разработанный прототип может использоваться для расширения линейки продуктов команды Acronis

5.1 Примеры полученных данных

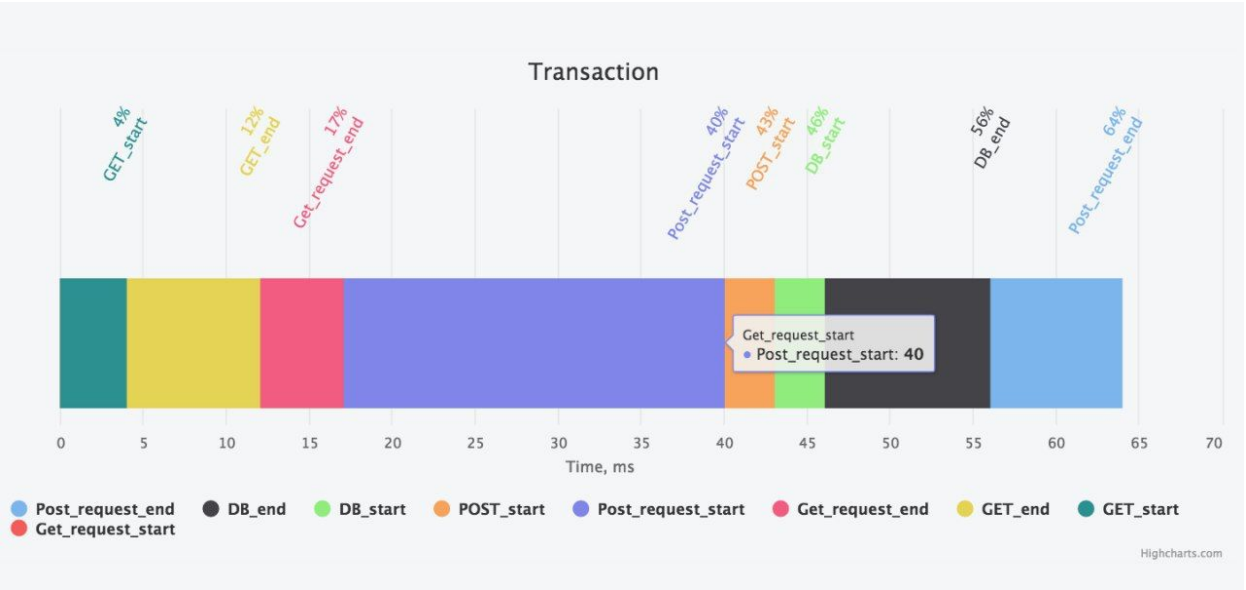

Рис. 9 Собранная транзакция (Приложение 8)

На Рис. 9 представлена собранная и обработанная транзакция со страницы /dbrequest веб-сервера. Большую часть транзакции составляет ожидание ввода ключа пользователем.

Рис. 10 График времени загрузки страницы (Приложение 9)

На Рис. 10 представлен график времени загрузки страниц. Min соответствует 5% самых быстрых загрузок, Max - 5% самых медленных.

Список литературы

1. https://www.frost.com/prod/servlet/press-release.pag?docid=291655896
   Frost & Sullivan, business consulting
2. HTTP: The Definitive Guide, *David Gourley* ISBN-13: 978-1565925090
3. *Miguel Grinberg*, Flask Web Development: Developing Web Applications with Python
4. Официальная документация Flask

   http://flask.pocoo.org/docs/0.12/
5. Стандарт HTML Navigation Timing

   https://www.w3.org/TR/navigation-timing/
6. Using SQLite, *Jay A. Kreibich*, ISBN 10:0-596-52118-9
7. UUID standard reference

   http://pubs.opengroup.org/onlinepubs/9629399/apdxa.htm

Приложения

Приложение 1. OPTIONS, GET, POST

OPTIONS

Используется для определения возможностей веб-сервера или параметров соединения для конкретного ресурса. В ответ серверу следует включить заголовок Allow со списком поддерживаемых методов. Также в заголовке ответа может включаться информация о поддерживаемых расширениях.

Предполагается, что запрос клиента может содержать тело сообщения для указания интересующих его сведений. Формат тела и порядок работы с ним в настоящий момент не определён; сервер пока должен его игнорировать. Аналогичная ситуация и с телом в ответе сервера.

Для того, чтобы узнать возможности всего сервера, клиент должен указать в URI звёздочку — «*». Запросы «OPTIONS * HTTP/1.1» могут также применяться для проверки работоспособности сервера (аналогично «пингованию») и тестирования на предмет поддержки сервером протокола HTTP версии 1.1.

Результат выполнения этого метода не кэшируется.

GET

Используется для запроса содержимого указанного ресурса. С помощью метода GET можно также начать какой-либо процесс. В этом случае в тело ответного сообщения следует включить информацию о ходе выполнения процесса.

Согласно стандарту HTTP, запросы типа GET считаются идемпотентными

Кроме обычного метода GET, различают ещё

Условный GET — содержит заголовки If-Modified-Since, If-Match, If-Range и подобные;

Частичный GET — содержит в запросе Range.

Порядок выполнения подобных запросов определён стандартами отдельно.

POST

Применяется для передачи пользовательских данных заданному ресурсу. Например, в блогах посетители обычно могут вводить свои комментарии к записям в HTML-форму, после чего они передаются серверу методом POST и он помещает их на страницу. При этом передаваемые данные (в примере с блогами — текст комментария) включаются в тело запроса. Аналогично с помощью метода POST обычно загружаются файлы на сервер.

В отличие от метода GET, метод POST не считается идемпотентным, то есть многократное повторение одних и тех же запросов POST может возвращать разные результаты (например, после каждой отправки комментария будет появляться очередная копия этого комментария).

При результате выполнения 200 (Ok) в тело ответа следует включить сообщение об итоге выполнения запроса. Если был создан ресурс, то серверу следует вернуть ответ 201 (Created) с указанием URI нового ресурса в заголовке Location.

Сообщение ответа сервера на выполнение метода POST не кэшируется.

Приложение 2. Обработка POST, GET запросов

```
@app.route('/workload', methods=['POST', 'GET'])
def workload():
    if request.method == 'POST':

        if not (request.form and 'min' in request.form):
            return 'Incorrect request'

        minutes = float(request.form['min'])
        _thread.start_new(calc_pi, (minutes * 60, ))
        return 'Started job for %f minutes' % minutes

    if request.method == 'GET':
        return render_template('workload.html', uuid_value=request.headers['uuid'])
```

Приложение 3. Инструментация flask

```
@app.before_request
def before_processing():
    starttime = str(time())
    request.headers = werkzeug.datastructures.Headers(request.headers)

    if request.form and 'uuid' in request.form:
        send_trans_part(request.form['uuid'], request.url, str(request.method)+'_start',
                        starttime)
        request.headers['uuid'] = request.form['uuid']

    elif request.json and 'uuid' in request.json:
        send_trans_part(request.json['uuid'], request.url, str(request.method) + '_start',
                        starttime)
        request.headers['uuid'] = request.json['uuid']

    else:
        curr_uuid = str(create_uuid())
        request.headers['uuid'] = curr_uuid
        send_trans_header(curr_uuid,request.headers['User-Agent'],
                          request.remote_addr)
        send_trans_part(curr_uuid, request.url, str(request.method)+'_start', starttime)
        print(request.headers)

@app.after_request
def after_processing(response):
    send_trans_part(request.headers['uuid'], request.url,
                    str(request.method) + '_end',  str(time()))
    return response
```

Приложение 4. Инструментация вызовов базы данных

```
@app.route('/dbrequest', methods=['POST', 'GET'])
def dbrequest():
    if request.method == 'POST':

        if not (request.form and 'item' in request.form):
            return 'Incorrect request'

        item = int(request.form['item'])
        my_uuid = request.headers['uuid']

        t1 = time()

        data_c.execute('SELECT value FROM data WHERE key=?', (int(item),))
        response = data_c.fetchone()

        result = time() - t1
        send_trans_part(my_uuid=my_uuid, my_time=result, page='/dbrequest',
                        info=('database_response,item='+str(item)))

        return render_template('dbresponse.html',
                        response=response[0] if response is not None else 'not found',
                                uuid_value=my_uuid)

    if request.method == 'GET':
        return render_template('dbrequest.html', uuid_value=request.headers['uuid'])
```

Приложение 5. Инструментация Javascript

```
<script type="text/javascript">

      var json = JSON.stringify({
         "uuid": "{{ uuid_value }}",
         "page": window.location.href,
         "timings": {
            "RequestStart":    window.performance.timing.requestStart,
            "RequestEnd":      window.performance.timing.responseEnd
         }
      });

      $.ajax({
      type: "POST",
      url: "http://52.59.88.40/api/v1.1/trans_part",
      dataType: "json",
      contentType: "application/json",
      data: json
   });

</script>
```

Приложение 6. Поддержка cross site scripting

```
@apm_receive_api.after_request
def some_callback(response):
    response.headers['Access-Control-Allow-Origin'] = '*'
    response.headers["Access-Control-Allow-Methods"] = 'POST'
    response.headers["Access-Control-Allow-Headers"] = 'Content-Type'
    response.headers["Access-Control-Max-Age"] = 1
    return response
```

Приложение 7. Нормализация транзакции из 4 временных отметок

```
def normalize_trans_four(work_set):

    outer_start = None
    outer_end = None
    inner_start = None
    inner_end = None

    uuid = work_set[0][0]

    for item in work_set:
        if item[2].find('Start') != -1:
            outer_start = item
        elif item[2].find('End') != -1:
            outer_end = item
        elif item[2].find('_start') != -1:
            inner_start = item
        elif item[2].find('_end') != -1:
            inner_end = item

    if outer_end is None:
        drop_trans(uuid)
        return 0

    trans_time = outer_end[3]/1000 - outer_start[3]/1000
    middle_time = (outer_end[3]/1000 - outer_start[3]/1000) - (inner_end[3] -
                                                                inner_start[3])
    half_middle_time = middle_time / 2

    new_inner_start = outer_start[3]/1000 + half_middle_time
    new_inner_end = outer_end[3]/1000 - half_middle_time

    data_c.execute('UPDATE trans_head SET summary_time = ? WHERE uuid =
?',
                   (str(trans_time), outer_end[0],))
    data_c.execute('UPDATE trans_part SET time = ? WHERE uuid = ?
                    AND time = ?',
                   (str(new_inner_start), uuid, str(inner_start[3])))
    data_c.execute('UPDATE trans_part SET time = ? WHERE uuid = ?
                    AND time = ?',
```

```
                    (str(new_inner_end),  uuid, str(inner_end[3])))
data_c.execute('UPDATE trans_part SET time = ? WHERE uuid = ?
                    AND time = ?',
                    (str(outer_end[3]/1000),  uuid, str(outer_end[3])))
data_c.execute('UPDATE trans_part SET time = ? WHERE uuid = ?
                    AND time = ?',
                    (str(outer_start[3]/1000), uuid, str(outer_start[3])))

return trans_time
```

Приложение 8. Генерирование графика транзакции

```
for i in range(0, item_count):
    i_for_series = i + series_correction

    length = (trans_parts[i][1] - previous_timestamp) / trans_time * 100
    previous_timestamp = trans_parts[i][1]
    series[i_for_series] = create_series_part(trans_parts[i][0], length)

    if not (db_item is None):
      if trans_parts[i][0].find('POST_start') != -1:
        series_correction = series_correction + 2

        window = trans_parts[i + 1][1] - previous_timestamp
        indent = (window - db_item) / 2
        print(window)
        print(indent)
        print(db_item)
        series[i + 1] = create_series_part('DB_start', (db_item) /
                                                    trans_time * 100)
        series[i + 2] = create_series_part('DB_end', indent / trans_time * 100)
        series[i]['data'][0] = indent / trans_time * 100

        previous_timestamp = trans_parts[i + 1][1] - indent
```

Приложение 9. Генерирование графика загрузки страницы

```
for item in final_trans_compl:
    place = int((item[2] - times[0]) // step)
    if place == PLOT_POINTS:
        place = place - 1
    if plot[place] is None:
        plot[place] = []

    plot[place].append(item[1])

sums = [None] * PLOT_POINTS

for i in range(0, PLOT_POINTS):
    if plot[i] is None:
        continue

    sums[i] = sum(plot[i]) / len(plot[i])

for i in range(0, PLOT_POINTS):
    if plot[i] is None:

        end = None
        for j in range(i, PLOT_POINTS):
            if not (plot[j] is None):
                end = j

        end_sum = sum(plot[end]) / len(plot[j])
        start_sum = sums[i-1]
        step = (end_sum - start_sum)/(end - i)

        for j in range(i, end):
            sums[j] = start_sum + j * step

        continue
```