GeekBrains

Урок 5. Анализ производительности, нагрузочное тестирование

На этом уроке:

- дадим определение архитектуре приложения в терминах общей теории систем (ОТС);
- рассмотрим элементы архитектуры, которые влияют на производительность приложения (bottleneck);
- изучим современные подходы и инструментарий для определения и наблюдения за производительностью приложения в реальном времени;
- выполним практическую работу: имитируем высокую нагрузку на приложение, определим пределы его производительности и проанализируем выявленные проблемы.

Оглавление

Введение

Под **производительностью приложения** нужно понимать набор количественных характеристик, которые позволяют оценить скорость и качество выполнения определенных операций. Требования к

производительности должны быть четко сформулированы еще на этапе проектирования приложения, ведь от этого зависит его будущая *архитектура*.

Для того, чтобы анализировать производительность приложений (Application Performance Analysis), необходимо:

- собрать данные для первоначального анализа архитектуры разрабатываемого приложения;
- своевременно выявлять сбои, ошибки и другие инциденты, влияющие на производительность приложения;
- выполнять автоматический мониторинг производительности ключевых функций.

Как указано в книге Documenting Software Architectures: Views and Beyond, **архитектура приложения** - это модель, структура, функциональность и взаимосвязь компонентов *информационной системы*.

Согласно стандарту ISO/IEC 2382:2015 **информационная система** - это *система*, предназначенная для хранения, поиска и обработки информации, и соответствующие организационные ресурсы, которые обеспечивают и распространяют информацию.

Система - это совокупность элементов произвольной природы, находящихся в отношениях и связях друг с другом, которая образует определенную целостность и характеризуется эмерджентностью. Как отмечал Густав Бергманн в своей статье Holism, Historicism, and Emergence в 1944 году, *эмерджентность* в теории систем - наличие у какой-либо системы особых свойств, не присущих её подсистемам и блокам, а также сумме элементов, не связанных особыми системообразующими связями.

Таким образом, *архитектура* приложения описывает набор элементов, из которых состоит приложение. Элементы приложения связаны между собой и образуют определенную структуру. За счет эффекта эмерджентности составные части приложения обогащают его, и у приложения появляются новые свойства, которых нет у отдельных элементов, входящих в его состав. Соответственно, у приложения появляется новая ценность. Характеристики связей между элементами, в частности: пропускная способность, число ошибок, задержки передачи данных - влияют на качество приложения. Падение значения характеристик ведет к потере качества приложения в целом и, как следствие, к падению его ценности.

Прежде чем перейти к теоретической части урока, введем некоторые уточнения. Под *приложением* будем понимать абстрактный сервис или набор сервисов. Сервис разворачивается на серверной инфраструктуре и доступен пользователям. Характеристики серверной инфраструктуры не имеют принципиального значения.

Теория урока

Анализ производительности приложения будем осуществлять, пользуясь терминологией ОТС - *система, надсистема, подсистема*. В рамках изучаемой темы, системой является сервис или набор

сервисов, надсистемой - пользователи сервиса, а подсистемой - серверная инфраструктура (см. Рисунок 1).

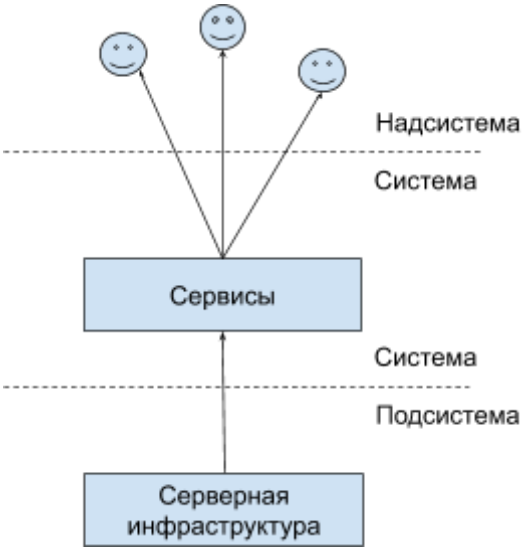

Рисунок 1

На рисунке 1 показаны надсистема, система, подсистема и связи между ними. Можно утверждать, что характеристики связей “надсистема-система” и “система-подсистема” определяют качество приложения. Если приложение представляет собой набор сервисов, взаимодействующих друг с другом, то искомые характеристики должны быть определены для каждого сервиса.

Чтобы определить потенциально проблемные места архитектурного решения необходимо выявить количественные метрики, характеризующие связи “надсистема-система” и “система-подсистема”. К наиболее часто возникающим проблемам, влияющим на производительность, можно отнести:

- ограничение производительности подсистемы влияет на доступность ресурсов подсистемы для системы;
- ограничение производительности системы для надсистемы влияет на доступность ресурсов системы для надсистемы - пользователей;
- собственные ограничения производительности надсистемы.

Ниже подробно рассмотрена природа возникновения указанных ограничений.

Производительность подсистемы

В отрыве от своих связей серверная инфраструктура также является системой. Для нас, разработчиков сетевых приложений, данная система состоит из операционной системы (ОС), вступающей во взаимодействие с подсистемами процессора, иерархической памяти (оперативная память и постоянная) и сетевого адаптера, с чем можно подробнее ознакомиться в книге Internetworking With TCP/IP. ОС является надсистемой рассматриваемых подсистем.

В 2012 Брендан Грегг, один из ведущих инженеров Netflix, предложил универсальную методологию анализа производительности серверных систем - USE-метод. Подробно с USE-методом можно ознакомиться в его статье. Кратко, Грегг охарактеризовал свой метод следующим правилом - для каждого элемента системы следует определить такие метрики, как *степень утилизации*, *степень насыщения* и *наличие ошибок*, где:

- **степень утилизации (utilization)** - процент загрузки элемента относительно максимальной допустимой;
- **степень насыщения или сатурация (saturation)** - количественная мера, показывающая сколько работы предстоит выполнить элементу. Например, число заявок в очереди: при увеличении числа заявок в очереди сатурация увеличивается, при уменьшении - уменьшается;
- **наличие ошибок (errors)** - число ошибочных срабатываний элемента.

Интересующие нас метрики часто определены в API ОС. Эрик Шрок, один из разработчиков ядра ОС Solaris, отмечает, что для ОС Linux все они находятся в файловой системе ProcFS. В таблице 1 представлены файлы ProcFS, содержащие мгновенные значения параметров рассматриваемых подсистем, которые можно использовать в качестве метрик.

> *Примечание*. Говоря об ОС далее будем иметь в виду ОС Linux, так как она фактически является современным промышленным стандартом для серверной инфраструктуры.

Таблица 1

	Утилизация	**Сатурация**	**Ошибки**
Процессор	/proc/stat	/proc/loadavg	?
Оперативная память	/proc/meminfo	/proc/vmstat	?
Постоянная память	/proc/diskstats	/proc/diskstats	?
Сеть	/proc/net/dev	/proc/net/	?

Источники метрик для определения наличия ошибок в подсистемах не указаны. Это не случайно. Дело в том, что не все USE-метрики определяются однозначно. Их определение зависит от предметной области и конфигурации системы.

Рассмотрим неоднозначность определения метрик на примере ошибок в сети. Для того, чтобы охарактеризовать число ошибок для HTTP-подсистемы логично использовать число кодов ответа “502” (Bad Gateway). Однако, этот же самый метод будет неприменим для TCP-подсистемы. Здесь в качестве метрики для оценки числа ошибок целесообразно использовать процент потерянных пакетов.

Закрепление знаний. Предложите свои варианты измерения числа ошибок для процессора, оперативной памяти, постоянной памяти. Обсудите их с однокурсниками, преподавателем. Обратите внимание, что единственно правильного варианта не существует.

Не все однозначно и с определением метрик для сатурации диска. Так для HDD пределом насыщения будет скорость вращения в об/мин, а для SSD диска ограничениями могут выступать технические характеристики не только самого диска, но и материнской платы. Аналогичные вопросы есть к утилизации и сатурации сетевого подключения.

Рассмотрим файл `/proc/stat`. Файл содержит мгновенные значения метрик утилизации для процессора. Чтобы оценить производительность CPU необходимо проанализировать данные файла. Очевидно, что заниматься сбором и анализом значений метрик вручную неудобно и нецелесообразно, так как мгновенные значения не позволяют оценить изменения значений параметров во времени, да и формат представления не предназначен для чтения инженером. Для анализа данных используются специальные приложения. Речь о них пойдет ниже.

```
# cat /proc/stat
cpu  1026 0 2443 978134 840 0 59 0 0 0
cpu0 499 0 1251 488249 438 0 33 0 0 0
cpu1 527 0 1191 489884 401 0 25 0 0 0
intr 1023604 37 0 0 72 463 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 58 0 14745 2795 48
1328 0 539 1 0 75385 80448 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
ctxt 1997811
btime 1604053605
processes 1834
procs_running 4
procs_blocked 0
softirq 425644 0 194432 44 558 4056 0 1 126464 0 100089
```

Закрепление знаний. Изучите данные файла `/proc/stat` (см. выше) и расскажите все что сможете о загрузке процессора. Необходимую справочную информацию можно найти в статье [How to read the Linux /proc/stat file](). В случае вопросов обратитесь к преподавателю.

Системы с TSDB

Мы столкнулись с проблемой и выяснили, что ручной анализ производительности систем практически невозможен из-за большого количества обрабатываемых данных и их машинно ориентированного представления. Для автоматизации анализа производительности систем разработаны специальные приложения - системы, использующие для хранения данных базы данных временных рядов ([Time Series Database](), TSDB).

Примером служит система *Prometheus*, которая была разработана в 2012 году на языке Go. Стоит отметить, что язык Go в те годы только начал набирать популярность и активно развиваться. Система Prometheus имеет полностью открытый исходный код и предоставляет десятки инструментов для анализа производительности приложений.

При помощи Prometheus можно проводить сбор данных одним из двух способов:

- **Метод "Pull"** - система собирает временные ряды с агентов, имеющих доступ к метрикам наблюдаемых систем, с заданной периодичностью. Метод подходит для сбора метрик приложений с активным сетевым интерфейсом и продолжительным временем жизни, таких как сервисы и демоны.
- **Метод "Push"** - агенты сами отправляют метрики системе по мере необходимости. Метод подходит для приложений, не имеющих слушающего сетевого интерфейса, или приложений с коротким временем жизни, таких как cron-задачи.

Агенты - это приложения, предназначенные для согласования форматов метрик наблюдаемой системы с форматами представления временных рядов в системе Prometheus. Например, для наблюдения за состоянием ProcFS (см. таблицу 1) был разработан стандартный агент `node_exporter`. Он читает файлы `/proc/*` и создает метрики, описывающие состояние сервера в формате системы Prometheus.

Например: `node_exporter` производит расчет метрики `node_cpu_seconds_total` - показатель утилизации процессора, который определяется по файлу `/proc/stats`. Данная метрика отображает время нахождения процессора в различных состояниях: `user`, `system`, `idle` и т.п.

Агенты Prometheus разработаны для балансировщиков, баз данных (реляционных и альтернативных), оркестраторов, систем виртуализации и других подсистем. Не стесняйтесь использовать эти готовые решения в работе, чтобы лучше понимать поведение подсистем, входящих в состав вашего программного продукта, и не только. Следует отметить, что агенты системы Prometheus являются сторонней взаимодействующей системой, которую можно использовать не только для сбора данных подсистем, но и систем, и надсистем.

Получить данные из системы Prometheus можно при помощи специального языка запросов временных рядов. Основные его функции приведены в документации.

Производительность системы

В предыдущем разделе мы рассмотрели способы диагностики и устранения проблем производительности на уровне подсистем. Перейдем на уровень выше - к системе. Использование метода USE на уровне системы нецелесообразно, т.к. система - это сервис, а не ресурс, как, например, процессор или оперативная память.

Брендан Грегг писал “Я разработал USE-метод, чтобы научить других быстро решать *общие* проблемы производительности”. Но проблемы производительности на уровне сервиса не являются общими. Они уникальны относительно исходного технического задания.

С уникальностью проблем производительности на уровне сервиса столкнулись и инженеры компании Google. В качестве решения они предложили метод мониторинга сервисов под названием “Четыре золотых сигнала”, который описали в 2016 году в книге SRE. К важным сигналам относятся:

- **трафик (requests)** - количество входящих запросов к сервису в единицу времени;
- **ошибки (errors)** - количество запросов вызвавших ошибочную ситуацию;
- **задержка (duration)** - время обработки запроса сервисом;
- **насыщение (saturation)** - степень близости сервиса к своей максимально возможной производительности.

В 2015 году, аббревиатура *RED* (*Requests Errors Duration),* составленная по первым буквам трех из четырех указанных метрик, уже упоминалась в презентации Тома Вилки, инженера Weave Works. Google предложили аббревиатуру REDS (*Requests Errors Duration Saturation)* на год позже. С развитием технологий автоматического масштабирования сервисов, буква “S” потеряла значимость и отмерла. Сегодня RED-метод является устоявшимся и принятым сообществом понятием. Метод упоминался на GrafanaCon EU в 2018 году. Компания Percona создала панели для мониторинга РСУБД MySQL с использованием этого подхода.

RED-метод можно применять для определения производительности HTTP-сервиса. В общем случае, для протокола HTTP в терминологии RED-метода будут важны:

- число HTTP-запросов со стороны клиента (*requests*);
- число HTTP-ответов с кодами множеств 4xx и 5xx (*errors*);
- время задержки HTTP-ответа на HTTP-запрос в сервисе (*duration*).

Опишем RED-метрики для оценки производительности абстрактной HTTP-системы с использованием стандартной библиотеки Prometheus для языка Go - *Prometheus Go client library*.

Важно! Не забудьте импортировать библиотеку.

```
var duration = prometheus.NewSummaryVec(
  prometheus.SummaryOpts{
     Name:       "duration_seconds",
     Help:       "Summary of request duration in seconds",
     Objectives: map[float64]float64{0.9: 0.01, 0.95: 0.005, 0.99: 0.001},
  },
  []string{labelHandler, labelMethod, labelStatus},
)

var errorsTotal = prometheus.NewCounterVec(
  prometheus.CounterOpts{
     Name: "errors_total",
     Help: "Total number of errors",
  },
  []string{labelHandler, labelMethod, labelStatus},
```

```
    )

    var requestsTotal = prometheus.NewCounterVec(
      prometheus.CounterOpts{
         Name: "request_total",
         Help: "Total number of requests",
      },
      []string{labelHandler, labelMethod},
    )
```

Оперируя переменными `duration`, `errorsTotal` и `requestsTotal` из примера выше и используя паттерн стратегии, создадим простейшую функцию-обертку для обработчика `http.HandlerFunc` HTTP-интерфейса нашего сервиса:

```
var MeasurableHandler = func(h http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
     t := time.Now()
     m := r.Method
     p := r.URL.Path

     requestsTotal.WithLabelValues(p, m).Inc()
     mw := newMeasurableWriter(w)
     h(mw, r)
     if mw.Status()/100 > 3 {
        errorsTotal.WithLabelValues(p, m, strconv.Itoa(mw.Status())).Inc()
     }
     duration.WithLabelValues(p, m,
strconv.Itoa(mw.Status())).Observe(time.Since(t).Seconds())
  }
}
```

> *Закрепление знаний.* Для кода выше обоснуйте наличие собственной имплементации интерфейса `http.ResponseWriter`, создаваемой функцией `newMeasurableWriter`. В качестве подсказки можете использовать исходный код вспомогательного пакета `promhttp`. Обсудите решение с однокурсниками и с преподавателем.

Для анализа производительности системы в произвольный момент времени, можно обернуть обработчики HTTP-запросов функцией `MeasurableHandler`. Использование же TSDB для хранения временных рядов позволяет проводить анализ на выборках данных за выбранный период.

До сих пор мы рассматривали систему как один сервис. Но в общем случае, система - это *набор сервисов* (приложений, РСУБД, распределенных очередей и т.д.), которые взаимодействуют друг с другом. В терминах ОТС их можно представить, как в*заимодействующие системы*.

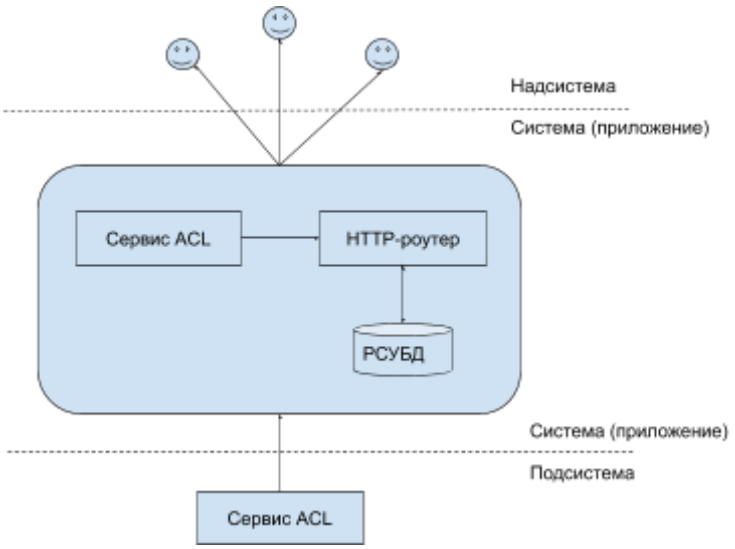

Рисунок 2

На рисунке 2 приведен пример приложения в составе которого группа сервисов:

- сервис HTTP-роутера;
- микро-сервис проверки прав доступа;
- БД некоторых сущностей.

Каждый сервис из состава приложения представляет собой отдельную систему. При этом серверная инфраструктура остается подсистемой для каждого сервиса, а пользователи приложения - надсистемой. Для пользователей же, приложение, которое является набором взаимодействующих систем как и раньше будет “черным ящиком”.

Важно! На рисунке 2 для каждого сервиса выделяется отдельная подсистема, то есть каждый сервис работает на отдельном сервере. Такое решение представляется наиболее целесообразным, но оно не является единственно возможным. Так, например, все сервисы приложения могут запускаться на одной серверной инфраструктуре или распределяться между несколькими серверами произвольным образом. Архитектура решения определяется требованиями технического задания. Подробнее об этом поговорим в блоке “Нагрузочное тестирование”

В рассматриваемом примере RED-метрик сервиса HTTP-роутера недостаточно для анализа и выявления проблем производительности системы. Так, например, если мы будем использовать только RED-метрики HTTP-роутера, то не сможем достоверно определить, что является причиной роста задержки получения ответов от него. Задержка может быть вызвана не только внутренними проблемами HTTP-роутера и его подсистем, но и проблемами *любого связанного с ним сервиса и его подсистем*. Поэтому для локализации проблемы производительности требуется добавить RED-метрики на связи HTTP- роутера со всеми системами приложения.

Производительность надсистемы

Итак, мы уже знаем, как обеспечивать контроль производительности серверной инфраструктуры и сервиса, а также удержать ключевые метрики производительности в заданных интервалах на уровне системы и подсистем. Но это не гарантирует удовлетворительную производительность на уровне надсистемы - конечного пользователя:

- надсистема является системой с собственными подсистемами, имеющими *конечные* вычислительные возможности. Важны размер устройства (карманное, планшет, стационарное), производитель и мощность процессора, объем и скорость работы оперативной памяти, вид сетевого соединения: от оптоволокна до GPRS, скорость передачи данных и стабильность подключения к провайдеру ISP. В общем случае, HTML-страница на мобильном телефоне с GPRS соединением отображается медленнее, чем на стационарном многопроцессорном ПК с выделенным оптоволоконным каналом к магистральному ISP.
- разнообразие пользовательских систем (ОС Android, Windows, iOS, OSX, Linux), множество браузеров (Chrome, Safari, Firefox, Samsung Internet, Edge, Opera и т.д), на порядок большее множество активных версий браузеров ведет к тому, что ответы сервера обрабатываются по различным алгоритмам, с различными качественными характеристиками.
- географическое местоположение элементов надсистемы относительно системы не менее важно. Инженер компании Microsoft Колинн Скотт в 2012 году опубликовал интерактивный актуализируемый сервис под названием “Величины задержек, которые должен знать каждый программист” - Latency Numbers Every Programmer Should Know. *Например:* если пользователь некоторого сервиса, расположенного в Канаде, находится в Нидерландах, к задержке ответа со стороны сервиса необходимо прибавлять ~150 мс, что, безусловно, влияет на итоговую производительность всей системы.

Проблемы производительности надсистемы, связанные с вычислительными возможностями пользовательских устройств и с разнообразием ОС и браузеров, решить на стороне сервера невозможно. Поэтому рассматривать их в рамках курса по серверному программированию мы не будем.

Анализ производительности географически-распределенных систем ничем не отличается от анализа производительности обычных систем и осуществляется RED-методом, как описано в разделе “Производительность системы”. С некоторыми серверными решениями проблемы географического распределения пользователей можно ознакомиться в статье.

Анализ производительности распределенной вычислительной системы

В этом блоке мы научимся:

- использовать систему Prometheus для сбора метрик системы;
- использовать средство визуализации Grafana для анализа производительности системы;
- проводить анализ производительности системы с заданной архитектурой (см. рисунок 2).

В ходе практической работы разработаем следующие сервисы:

- сервис обработки запросов (сервис HTTP-роутера);
- сервис контроля прав доступа (сервис ACL);
- сервис хранения данных (СУБД).

Для упрощения задачи из инфраструктуры системы исключены БД прав доступа и кэш, представленные на рисунке 2. При желании, после выполнения практической работы вы можете самостоятельно расширить архитектуру системы, дописать недостающие сервисы и провести анализ производительности.

Сервис обработки запросов

Сервис HTTP-роутера является внешним интерфейсом системы и обрабатывает два запроса - добавление в БД новой сущности и возврат всех сущностей.

`AddEntityHandler` - метод добавления новой сущности в БД. При добавлении новой сущности, сервис обработки запросов сверяется с микросервисом контроля прав доступа. Если последний вернул подтверждение актуальности авторизационного токена, сервис добавляет новую сущность в РСУБД, в противном случае он возвращает ошибку.

`ListEntitiesHandler` - метод возврата списка добавленных сущностей. Осуществляет их возврат по запросу любого пользователя.

Важно! При разработке сервиса обработки запросов для наблюдения за производительностью системы используется обработчик `MeasurableHandler`. Ознакомиться с ним можно в разделе “Производительность системы” теоретической части урока.

Пример исходного кода сервиса приведен ниже:

```
var (
  db *sql.DB

  measurable = red.MeasurableHandler

  router = mux.NewRouter()
```

```
  web    = http.Server{
     Handler: router,
  }
)

func init() {
  router.
     HandleFunc("/entities", measurable(ListEntitiesHandler)).
     Methods(http.MethodGet)
  router.
     HandleFunc("/entity", measurable(AddEntityHandler)).
     Methods(http.MethodPost)

  var err error
  db, err = sql.Open("mysql", "root:test@tcp(mysql:3306)/test")
  if err != nil {
     panic(err)
  }
}

func main() {
  go func() {
     http.Handle("/metrics", promhttp.Handler())
     if err := http.ListenAndServe(":9090", nil); err != http.ErrServerClosed {
        panic(fmt.Errorf("error on listen and serve: %v", err))
     }
  }()
  if err := web.ListenAndServe(); err != http.ErrServerClosed {
     panic(fmt.Errorf("error on listen and serve: %v", err))
  }
}

const sqlInsertEntity = `
      INSERT INTO entities(id, data) VALUES (?, ?)
`

func AddEntityHandler(w http.ResponseWriter, r *http.Request) {
  res, err := http.Get(fmt.Sprintf("http://acl/identity?token=%s",
r.FormValue("token")))
  switch {
  case err != nil:
     w.WriteHeader(http.StatusServiceUnavailable)
     return
  case res.StatusCode != http.StatusOK:
     w.WriteHeader(http.StatusForbidden)
     return
  }
  res.Body.Close()

  _, err = db.Exec(sqlInsertEntity, r.FormValue("id"), r.FormValue("data"))
  if err != nil {
     w.WriteHeader(http.StatusInternalServerError)
     return
  }
}

const sqlSelectEntities = `
      SELECT id, data FROM entities
`
```

```
type ListEntityItemResponse struct {
  Id string   `json:"id"`
  Data string `json:"data"`
}

func ListEntitiesHandler(w http.ResponseWriter, r *http.Request) {
  rr, err := db.Query(sqlSelectEntities)
  if err != nil {
     w.WriteHeader(http.StatusInternalServerError)
     return
  }
  defer rr.Close()

  var ii = []*ListEntityItemResponse{}
  for rr.Next() {
     i := &ListEntityItemResponse{}
     err = rr.Scan(&i.Id, &i.Data)
     if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        return
     }
     ii = append(ii, i)
  }
  bb, err := json.Marshal(ii)
  if err != nil {
     w.WriteHeader(http.StatusInternalServerError)
     return
  }
  w.Header().Add("Content-Type", "application/json")
  _, err = w.Write(bb)
  if err != nil {
     w.WriteHeader(http.StatusInternalServerError)
     return
  }
}
```

Сервис контроля прав доступа

Микросервис контроля прав доступа представляет собой HTTP обработчик, который проверяет соответствие GET-параметров токена на удовлетворение внутренним требованиям по ограничению доступа. Если проверка прошла успешно обработчик возвращает HTTP код 200, в противном случае - HTTP код 401.

Важно! При разработке сервиса контроля прав доступа для наблюдения за производительностью системы используется обработчик `MeasurableHandler`. Ознакомиться с ним можно в разделе “Производительность системы” теоретической части урока.

Пример исходного кода микросервиса контроля прав доступа приведен ниже:

```
var (
  measurable = red.MeasurableHandler

  router = mux.NewRouter()
  web    = http.Server{
```

```
    Handler: router,
  }
)

func init() {
  router.
    HandleFunc("/identity", measurable(GetIdentityHandler)).
    Methods(http.MethodGet)
}

func main() {
  go func() {
    http.Handle("/metrics", promhttp.Handler())
    if err := http.ListenAndServe(":9090", nil); err != http.ErrServerClosed {
      panic(fmt.Errorf("error on listen and serve: %v", err))
    }
  }()
  if err := web.ListenAndServe(); err != http.ErrServerClosed {
    panic(fmt.Errorf("error on listen and serve: %v", err))
  }
}

func GetIdentityHandler(w http.ResponseWriter, r *http.Request) {
  if r.FormValue("token") == "admin_secret_token" {
    w.WriteHeader(http.StatusOK)
    return
  }
  w.WriteHeader(http.StatusUnauthorized)
}
```

Сервис хранения данных

Сервис хранения данных является важной частью архитектуры системы. Для реализации сервиса будем использовать РСУБД MySQL, выступающую в качестве персистентного хранилища сущностей. Пример описания таблицы хранения сущностей на MySQL приведен ниже.

```
CREATE TABLE IF NOT EXISTS entities (
   id INT PRIMARY KEY,
   data VARCHAR(32)
);
```

Выполнить данный код DDL возможно с использованием утилиты `mysql`

Описание инфраструктуры

Опишем архитектуру исследуемой системы композицией сервисов:

```
version: "3.7"
services:

 acl:
   build:
     dockerfile: Dockerfile
     context: .
```

```
    command: "./bin/acl"
    ports:
      - 8001:80

  router:
    build:
      dockerfile: Dockerfile
      context: .
    command: "./bin/router"
    ports:
      - 8002:80

  mysql:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: test
      MYSQL_DATABASE: test
    ports:
      - 8010:3306

  grafana:
    image: grafana/grafana:6.4.4
    environment:
      - "GF_SECURITY_ADMIN_USER=admin"
      - "GF_SECURITY_ADMIN_PASSWORD=password"
    ports:
      - 3000:3000

  node-exporter:
    image: prom/node-exporter:v1.0.1
    ports:
      - 9100:9100

  prometheus:
    image: prom/prometheus:v2.22.0
    volumes:
      - "./prometheus.yml:/etc/prometheus/prometheus.yml:ro"
    ports:
      - 9090:9090
```

`acl`, `router`, `mysql` описывают приложение. Для наблюдения за производительностью подсистемы, в качестве агента представлен `node-exporter`. Система `prometheus` предназначена для сбора метрик методом “Pull” и хранения их в TSDB. Сервис для графического отображения метрик `grafana` предоставит нам возможность наблюдать за поведением системы и подсистемы в реальном времени с исторической перспективой

Конфигурация Prometheus

Сконфигурируем систему Prometheus для сбора и хранения данных:

```
global:
 scrape_interval: 5s
rule_files:
scrape_configs:
 - job_name: 'acl'
   static_configs:
```

```
    - targets: ['acl:9090']
- job_name: 'router'
  static_configs:
    - targets: ['router:9090']
- job_name: 'node-exporter'
  static_configs:
    - targets: ['host.docker.internal:9100']
```

Файл конфигурации системы Prometheus – `prometheus.yml`. В таблице 2 приведено описание параметров конфигурации.

Таблица 2

Параметр	**Описание**	**Значение**
Секция global		
`scrape_interval`	Интервал сбора метрик	Каждые 5 секунд
Секция scrape_configs		

`job_name`	Наименование сборщика метрик	`acl`
		`router`
		`node-exporter`

`static_configs`	Указание статической конфигурации сборщика метрик	-

`targets`	Адрес через который агент возвращает метрики	`acl:9090`
		`router:9090`
		`host.docker.internal:9100`

Запуск инфраструктуры

Запуск осуществляется командой:

```
docker-compose -f docker-compose.yml up -d
```

Важно! Запустите инфраструктуру и убедитесь, что все сервисы запущены и работоспособны, используя команду `docker ps`.

Настройка визуализатора Grafana

Для настройки визуализатора графана нужно посетить http://localhost:3000/login и ввести данные, определенные в переменных `GF_SECURITY_ADMIN_USER` и `GF_SECURITY_ADMIN_PASSWORD` раздела “Описание инфраструктуры”. После чего в настройках нужно добавить новый источник данных типа “Prometheus”, указав в качестве URL его внутренний адрес: http://prometheus:9090

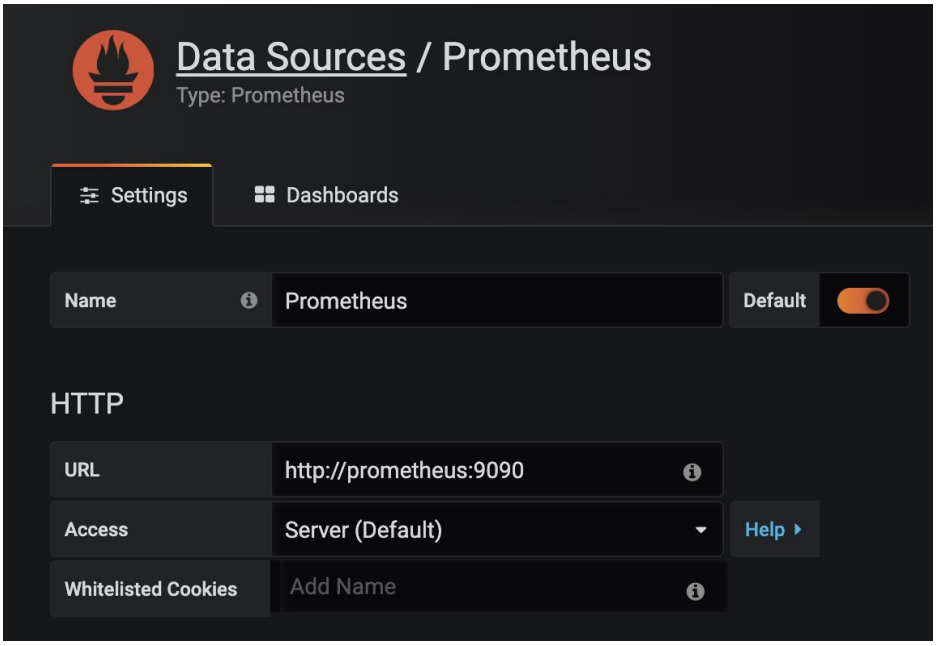

Рисунок 3

USE-метрики подсистемы

Получим USE-метрики - показания по утилизации и сатурации, - для работающей серверной инфраструктуры. Агент `node-exporter` производит чтение показаний из ProcFS, а Prometheus собирает данные в собственную TSDB с периодом, указанным в файле конфигурации `prometheus.yml` (в нашем случае 5 секунд). Доступ к TSDB осуществляется через web-интерфейс по адресу http://localhost:9090/graph. Web-интерфейс предлагает строку быстрого поиска, использующую язык запросов временных рядов.

В таблице 3 приведены запросы к системе Prometheus на языке временных рядов для получения USE-метрик подсистем - процессора, оперативной памяти, жесткого диска и сети.

Важно! В таблице 3 приведены примеры запросов, но ни в коем случае не единственно верный вариант получения USE-метрик.

Таблица 3

	Формула утилизации	**Формула сатурации**
Процессор	`1 - avg(irate(node_cpu_seconds_total{job="node-exporter",mode="idle"}[1m]))`	`sum(node_load1{job="node-exporter"})`
Память	`1 - sum(node_memory_MemFree_bytes{job="node-exporter"} + node_memory_Cached_bytes{job="node-exporter"} + node_memory_Buffers_bytes{job=`	`sum(rate(node_vmstat_pgpgin{job="node-exporter"}[1m]) + rate(node_vmstat_pgpgout{job="node-exporter"}[1m]))`

	`"node-exporter"}) / sum(node_memory_MemTotal_bytes {job="node-exporter"})`	
Диск	`1 - max(node_filesystem_avail_bytes{job="node-exporter"}) / max(node_filesystem_size_bytes {job="node-exporter"})`	`sum(rate(node_disk_reads_completed_total[1m]) + rate(node_disk_writes_completed_total[1m]))`

Сеть	`sum(rate(node_network_receive_bytes_total{job="node-exporter",device!="lo"}[1m]))`
	
	`sum(rate(node_network_transmit_bytes_total{job="node-exporter",device!="lo"}[1m]))`
		`sum(rate(node_network_receive_drop_total{job="node-exporter",device!="lo"}[1m]))`
		
		`sum(rate(node_network_transmit_drop_total{job="node-exporter",device!="lo"}[1m]))`

Если перенести формулы из таблицы 3 в сервис визуализатора Grafana, USE-метрики будут отображаться в виде графиков (рисунок 4), что позволит осуществлять мониторинг текущего состояния серверной инфраструктуры в режиме реального времени.

Рисунок 4

RED-метрики сервисов

После запуска разработанных сервисов наблюдение за поведением системы осуществляется при помощи стандартного HTTP-обработчика `/metrics`. Обработчик создается автоматически при запуске библиотеки [Prometheus Go client library](#).

В таблице 4 приведены формулы для получения RED-метрик системы на языке временных рядов.

Таблица 4

	Requests	Errors	Duration

acl.Identity	`rate(request_total{handler="/identity",job="acl",method="GET"}[1m])`	`rate(errors_total{handler="/identity",job="acl",method="GET"}[1m])`	`duration_seconds{handler="/identity",job="acl",method="GET",status="200",quantile="0.99"}`
			
			`duration_seconds{handler="/identity",job="acl",method="GET",status="200",quantile="0.95"}`
			
			`duration_seconds{handler="/identity",job="acl",method="GET",status="200",quantile="0.9"}`

router.addEntity	`rate(request_total{handler="/entity",job="router",method="POST"}[1m])`	`rate(errors_total{handler="/entity",job="router",method="POST"}[1m])`	`duration_seconds{handler="/entity",job="router",method="POST",status="200",quantile="0.99"}`
			
			`duration_seconds{handler="/entity",job="router",method="POST",status="200",quantile="0.95"}`
			
			`duration_seconds{handler="/entity",job="router",method="POST",status="200",quantile="0.9"}`

rounter.listEntities	`rate(request_total{handler="/entities",job="router",method="GET"}[1m])`	`rate(errors_total{handler="/entities",job="router",method="GET"}[1m])`	`duration_seconds{job="router",method="GET",status="200",quantile="0.99"}`
			
			`duration_seconds{job="router",method="GET",status="200",quantile="0.95"}`
			
			`duration_seconds{job="router",method="GET",status="200",quantile="0.9"}`

Обратите внимание на то, что для определения задержки (Duration) в таблице 4 приведены целых три метрики, а не одна, как для запросов (Requests) и ошибок (Errors). Значение временного ряда -

это случайная величина и анализировать ее целесообразно, используя существующий статистический аппарат.

Рассмотрим необходимость введения нескольких метрик для определения задержки на конкретном примере. Пусть на заводе работают 10 человек. Девять из них получают зарплату 100 рублей, а один - 1000 рублей. Как понять, насколько финансово благополучны сотрудники? Если рассчитать среднюю зарплату, получим что каждый сотрудник получает по 190 рублей. Но в действительности 90% рабочих таких денег никогда не видят. Если же рассчитать медианное значение, то среднее значение как раз таки составит 100 рублей. Для рассматриваемого примера эта оценка более правдива и лучше описывает ситуацию.

Вернемся к способам оценки задержки ответов сервиса. Анализировать все временные ряды, которые создает сервис, обрабатывающий, например, 100,000 и более запросов/секунду бессмысленно, т.к. это потребует слишком много ресурсов. Медианное значение в данном случае также не представляет ценности, ведь 50% запросов будут обрабатываться медленнее. Поэтому при анализе задержки сервисов принято рассматривать более высокие перцентили, чем медианный: 90-й (p.9), 95-й (p.95), 99-й (p.99), 99.99-й (p.9999). В таблице 4 приведены формулы для расчета задержки для перцентилей p.99, p.95 и p.9. Выбор перцентиля обычно определяется требованиями надсистемы.

Визуализируем RED-метрики в виде графиков в сервисе Grafana (рисунок 5):

Рисунок 5

Нагрузочное тестирование

Нагрузочное тестирование (load testing) - тестирование производительности, сбор показателей и определение производительности и времени отклика системы в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе. В процессе

нагрузочного тестирования моделируется ожидаемая в системе нагрузка и осуществляется наблюдение за показателями производительности и их анализ.

Для проведения нагрузочного теста воспользуется утилитой `wrk`. Утилита отправляет тестовые запросы на URL-адрес в течение заданного промежутка времени, используя указанное число тредов и соединений.

Настраиваемые параметры:

- число тредов (потоков);
- число соединений;
- продолжительность теста;
- допустимый таймаут на запросы;
- номер порта, который слушает приложение, или URL-адрес.

Внешний интерфейс системы содержит только два метода - `router.addEntity`, `router.listEntities`. Следовательно понадобятся всего два теста:

1. Тест запросов вида `` `GET /entities` ``;
2. Тест запросов вида `` `POST /entity?token=&id=&data=` ``.

Нагрузочный тест для запросов `` `GET /entities` `` имеет вид:

```
wrk -t1 -c1 -d 5m http://localhost:8002/entities
```

> **Важно!** Попробуйте запускать утилиту `wrk` с различными параметрами. Оцените, как значения параметров влияют на результаты теста.

Разработать нагрузочный тест для запросов `` `POST /entity?token=&id=&data=` `` чуть сложнее - необходимо опробовать различные типы данных добавляемых сущностей и проверить возврат ошибок. Для подобных кейсов утилита `wrk` имеет поддержку скриптового языка Lua.

Ниже приведен пример простейшего скрипта запроса:

```
math.randomseed(os.time())
request = function()
  local k = math.random(0, 1000)
  local t
  if k > 950 then
     t = "incorrect_admin_token"
  else
     t = "admin_secret_token"
  end

  local url = "/entity?token="..t.."&id="..k.."&data="..k
  return wrk.format("POST", url)
end
```

Скрипт создает случайные данные для добавления и в 5% запросов использует неверный токен доступа. Команда запуска его чуть сложнее:

```
wrk -t1 -c1 -d5m -s ./test/wrk.lua http://localhost:8002
```

В идеале тестирующая и тестируемая инфраструктуры должны быть разнесены по разным серверам, чтобы исключить возможность использования одной и той же подсистемы для приложения и для теста и ложные корреляции. Ведь тест - это тоже программа, потребляющая ресурсы. Если ресурсы общие и для теста и для наблюдаемой системы, возможна ситуация, когда тест будет потреблять ресурсы, необходимые для работы наблюдаемой системы. Система начнет испытывать их дефицит из-за деградации связей с подсистемой и качество работы приложения снизится, что будет заметно на RED-метриках.

Установим число тредов и число соединения равными единице и запустим разработанные тесты на пять минут :

```
$: wrk -t1 -c1 -d 5m http://localhost:8002/entities

Running 5m test @ http://localhost:8002/entities
  1 threads and 1 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     8.92ms   13.87ms 468.49ms   95.43%
    Req/Sec   140.35     43.55   282.00     74.55%
  41834 requests in 5.00m, 0.94GB read
Requests/sec:    139.44
Transfer/sec:      3.20MB
```

Приложение обработало **41834** запроса на получение сущностей со скоростью **~140 запросов/секунду (rps)** со средней задержкой в **~9 ms**, максимальная задержка составляла **468 ms**.

```
$: wrk -t1 -c1 -d5m -s ./test/wrk.lua http://localhost:8002

Running 5m test @ http://localhost:8002
  1 threads and 1 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     8.39ms   14.81ms 462.92ms   95.58%
    Req/Sec   157.41     48.40   287.00     74.59%
  46879 requests in 5.00m, 4.16MB read
  Non-2xx or 3xx responses: 45928
Requests/sec:    156.24
Transfer/sec:     14.19KB
```

На добавление данных было отправлено **46879** запросов со скоростью **~156 rps**. Среднее время ответа сервиса **~8 ms**. На **45928** запроса приложение вернуло ошибку, что не удивительно, т.к. мы использовали максимум 1000 случайных сущностей для вставки.

Ниже приведены результаты тестов в виде графиков в Grafana.

Рисунок 6

По RED-метрикам справа видно, что поток запросов обрабатывался стабильно: не было ни резких скачков, ни падений. Число ошибок обработчика `acl.Identity` соответствует ожидаемым 5% от общего числа запросов. Задержка **p.99** обработки запросов стабильна. Обработчик `router.listEntities` не возвращал ошибок, что подтверждается показаниями надсистемы - утилиты тестирования `wrk`. Число ошибок `router.addEntity` ожидаемо велико.

Задержки **p.95** и **p.9** у всех обработчиков почти в 2 раза ниже показателя **p.99** и находятся близко друг к другу, что подтверждает целесообразность выбора языка Go для разработки высоконагруженных сервисов.

По USE-метрикам слева видно, что утилизация процессора с момента начала теста достигла **100%**. Ресурсов процессора было недостаточно. Оперативная память утилизировалась стабильно на уровне **34.5%**, что говорит об отсутствии явных утечек. В момент запуска теста видна сатурация по памяти, которая позже стабилизировалась, в виду адаптации системы идущему рейту запросов в **~300 rps**. Диск в процентном отношении не использовался. Наблюдаемая небольшая сатурация объясняется работой персистентного хранилища данных РСУБД MySQL - записи сопровождались обязательным вызовом `fsync`.

Учитывая сатурацию процессора на время теста в районе **5-6 единиц** для двухъядерной машины, можно сделать вывод, что для обработки подобного рейта запросов, количество ядер процессора должно быть в 3 раза выше текущего. Все остальные подсистемы не являлись узкими местами решения.

Таким образом, можно сделать вывод, что разработанная система способна обрабатывать до **~150 rps** в текущей конфигурации. При этом мы определили узкое место (bottleneck) нашего решения - это

процессор. Утилизация процессора и его сатурация во время теста выходили за значения, допустимые для комфортной обработки запросов.

Практические задания

Для закрепления полученных навыков выполните задания.

1. Измените нагрузочный тест для `router.addEntity` таким образом, чтобы число ошибок было менее 1%. Оцените изменений нагрузочного профиля.
2. Создайте набор RED-метрик для методов `Exec`, `Query` и `QueryRow` структуры `sql.Db` пакета `"database/sql"` , используя библиотеку системы Prometheus.
3. Предложите решение для обеспечения контроля производительности подсистемы сервиса хранения данных, используя возможности Percona PMM.
4. Попытайтесь самостоятельно расширить архитектуру системы, дописать некоторые недостающие сервисы и провести анализ производительности. В качестве дополнительного сервиса может выступать СУБД для хранения прав ACL - ранее мы описали права доступа непосредственно в коде функции `acl.Identity`.

Используемые источники

1. Книга по дизайну сервисных архитектур: Documenting Software Architectures: Views and Beyond;
2. Философский трактат Густава Бергмана об эмерджентности систем: Holism, Historicism, and Emergence;
3. Настольная книга разработчика сетевых приложений Internetworking With TCP/IP;
4. The USE Method, которым мы пользовались для определения производительности подсистем;
5. Исторический ревью о различиях ProcFS в разных ОС A brief history of /proc;
6. Статья How to read the Linux /proc/stat file может научить определять загруженность процессора не хуже чем это делают утилиты `top` и `htop`;
7. Вход в “кроличью нору” по Time series database;
8. Узнать особенности реализации `node_exporter` можно в репозитории;
9. Подробное руководство по языку запросов временных рядов Prometheus “Querying Prometheus”;

10. Книга Google SRE - это настольная библия разработчика высоконагруженных приложений;
11. Увлекательная презентация по анализу производительности микросервисов в реальном времени Monitoring Microservices;
12. В статье RED Method for MySQL Performance Analyses можно ознакомиться с принципами наблюдения за производительностью MySQL;
13. Репозиторий библиотеки Prometheus Go client library может быть интересен программистам, изучающим Go;
14. Сайт StatCounter раскрывает пропорции использования пользовательских браузеров и устройств в мире;
15. Значения, которые обязан знать каждый программист приведены на сайте;
16. В этой статье можно узнать как Google оперирует географически-распределенными данными;
17. Данная статья подскажет, как запустить Percona PMM, используя docker;
18. Тем, кто уже уверенно освоил Go и ищет что-то новое, может помочь Lua: reference manual.