GeekBrains

Архитектурный шаблон: брокер сообщений

На этом уроке:

- Дадим определение архитектурному шаблону брокера сообщений в терминах теории массового обслуживания
- Рассмотрим пример простейшего брокера сообщений
- Сравним существующие брокеры сообщений с открытым исходным кодом: Apache Kafka, RabbitMQ, NATS
- Выполним практическую работу: реализуем сервис оценок, аналогичный emojicom

Оглавление

Введение

Артур Эйсмонт, инженер компании Yahoo, в своей книге Web Scalability for Startup Engineers указал, что **брокер сообщений** - это архитектурный шаблон построения масштабируемой вычислительной системы, который обеспечивает доставку сообщений от источников приемникам. Понятие брокера сообщений обладает некоторой двойственностью. По данным глоссария IT-терминологии компании Gartner, брокер сообщений - это готовое приложение. Но с созидательной стороны вопроса - это всё-таки шаблон, который может использоваться для организации коммуникаций в разрабатываемом приложении

Сообщение - это просто объект некоторой структуры данных, который должен быть доставлен от источника к приемникам. В процессе доставки сообщения брокер может: изменить сообщение; заменить сообщение его проекцией; разделить его на отдельные сообщения; агрегировать несколько сообщений в одно и т.д. Также брокер решает, каким приемникам будет доставлено сообщение и как именно:

- Сообщение может быть доставлено как минимум один раз - *at least once*
- Максимум один раз - *at most once*
- Только один раз - *exactly once*

Идея построения приложений в виде системы, выполняющей передачу сообщений, уходит корнями в 1909 год, когда Агнер Крарруп Эрланг опубликовал статью The Theory of Probabilities and Telephone Conversations, основав Теорию массового обслуживания (ТМО). ТМО предлагает математический аппарат для анализа системы массового обслуживания (СМО) и определяет классификацию последних:

- СМО с потерями теряют сообщение, если его невозможно доставить
- СМО с ожиданием используют очереди бесконечной ёмкости для хранения сообщений, которые невозможно доставить
- СМО с ожиданием и ограничениями используют очереди конечной ёмкости для хранения сообщений, которые невозможно доставить. При достижении размера очереди предельной ёмкости, сообщения теряются

Представляя приложения в виде СМО, можно использовать аналитический аппарат ТМО с для анализа поведения приложения под *высокой нагрузкой* (highload). Рассмотрим подробнее основы данной теории.

Теория урока

Для упрощения, СМО принято представлять в виде “черного ящика” (black box). Данный подход является универсальным, так как не требует понимания принципов работы системы. Рассматриваются только процессы на входе и на выходе.

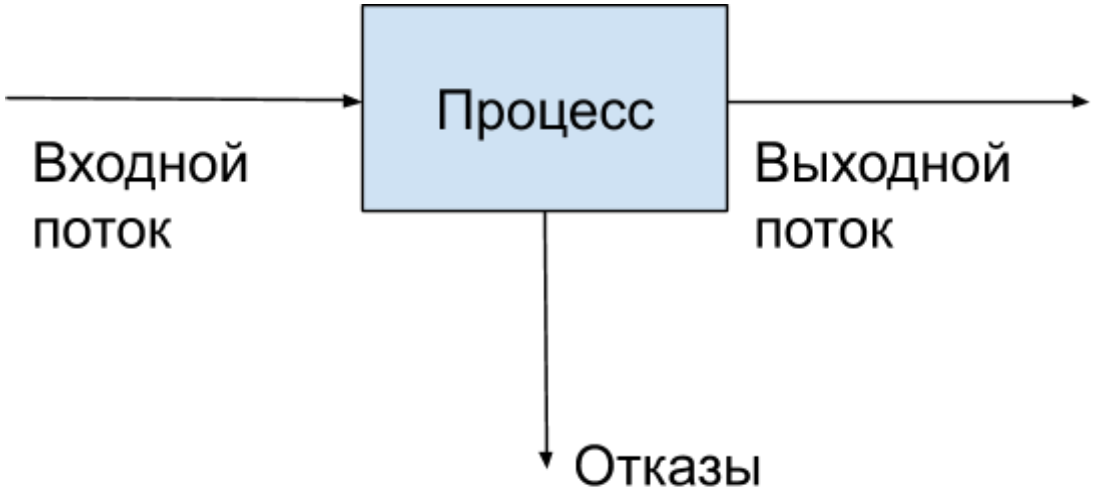


Рисунок 1

В рамках ТМО принято оперировать понятием *случайного процесса*. **Случайный процесс** - это последовательность случайных величин. Для СМО случайной величиной является сообщение. Рассмотрим схему на рисунке 1 подробнее:

- Входной поток - это случайный процесс, описывающий поступление сообщений в систему

- Процесс обслуживания - это случайный процесс обработки поступающих сообщений
- Поток отказов - это случайный процесс показывающий отказы при обработке поступающих сообщений
- Выходной поток - это случайный процесс, описывающий вывод сообщений из системы в результате их успешной обработки

Рассмотрим пример, применительно к текущему курсу: HTTP-сервер в качестве СМО. Тогда случайной величиной будет являться HTTP-запрос, а входной поток - это последовательность HTTP-запросов со стороны клиента с течением времени

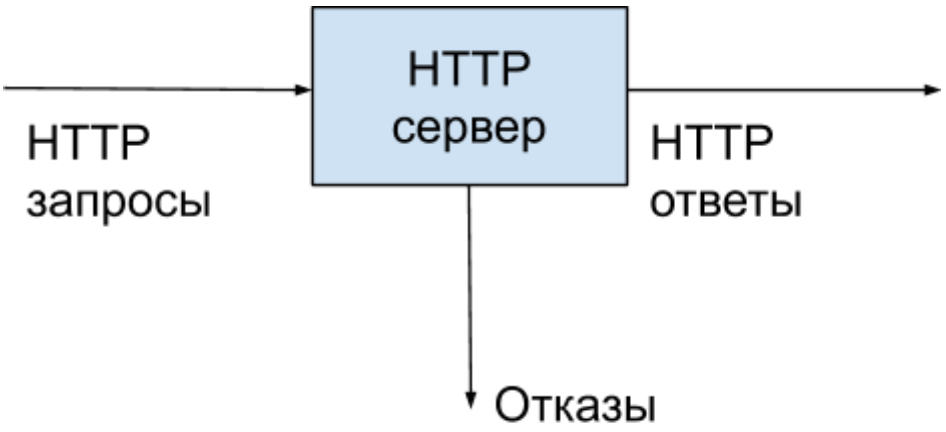


Рисунок 2

Выходным потоком на рисунке 2 будет являться последовательность HTTP-ответов. Если HTTP-сервер не может обработать запрос будет формироваться поток отказов:

- Невозможно подсоединиться к серверу `CURLE_COULDNT_CONNECT`
- Невозможно дождаться ответа `CURLE_OPERATION_TIMEDOUT`
- Ответ пуст `CURLE_GOT_NOTHING`
- Ошибка на отправлении данных `CURLE_SEND_ERROR`
- Ошибка при получении данных `CURLE_RECV_ERROR`

  Для самостоятельного изучения. В списке приведены коды ошибок популярной библиотеки libcurl, созданной ещё в 1996 году для выполнения HTTP-запросов. В настоящее время она поддерживает множество протоколов и, фактически является, промышленным стандартом. Предлагаю ознакомиться с полным списком возможных ошибок.

Если в качестве примера рассмотреть не HTTP-сервер, а HTTP-сервис, то в потоком отказов могут быть и корректные ответы с точки зрения протокола HTTP - ответы с кодами в диапазоне 400-499 - клиентские ошибки, и 500-599 - серверные ошибки.

Анализ систем массового обслуживания

Мы рассмотрели возможность использования математического аппарата ТМО для анализа СМО. Нашей целью является выполнение анализа путем определения выходного потока и потока отказов СМО при заданном входном потоке и процессе обслуживания. Такой вид СМО принято описывать формулой (1)

$$A/B/n \quad (1)$$

, где A - входной поток, B - процесс обслуживания, а n - число обработчиков

Как было указано ранее и входной поток, и процесс обслуживания являются случайными процессами. Большой вклад в изучение случайных процессов внес российский и советский математик Андрей Николаевич Колмогоров, исследовавший многие их них. Случайные процессы принято обозначать буквами: M - экспоненциальный случайный процесс; E - распределение Эрланга; H - гиперэкспоненциальное распределение; U - равномерное распределение и т.д. В рамках нашего курса мы не будем рассматривать каждых из них. Но стоит отметить факт наличия большого числа уже изученных случайных процессов.

На практике приходится сталкиваться с реальными процессами, поведение которые приближено к тем или иным формально описанным и изученным случайным процессам

Рисунок 3

На рисунке 3 представлен входной процесс некоторого приложения, работающего на продакшене. По внешнему виду можно определить, что это синусоида со случайной амплитудой и случайной частотой. Этот процесс можно записать формулой (2) и анализировать математически

$$S(t) = A * sin(\omega * t + \varphi) + B \quad (2)$$

Если распределение неизвестно, используется обозначение G - распределение общего вида. Распределение постоянной величины описывается символом D. Входной поток D - это уже не

случайный процесс, а постоянный. Процесс обслуживания D - это процесс, который тратит на обработку сообщения всегда одно и то же время

Вернемся к примеру HTTP-сервиса как СМО. Пусть HTTP-сервис имеет один обработчик, выполняющий сложение двух операндов (рисунок 4). Как было указано ранее, входным потоком сообщений являются HTTP-запросы. Мы не знаем, когда придет HTTP-запрос, в каком количестве они поступят, какова будет плотность потока запросов с течением времени - может быть, как на рисунке 3, а может и по-другому.

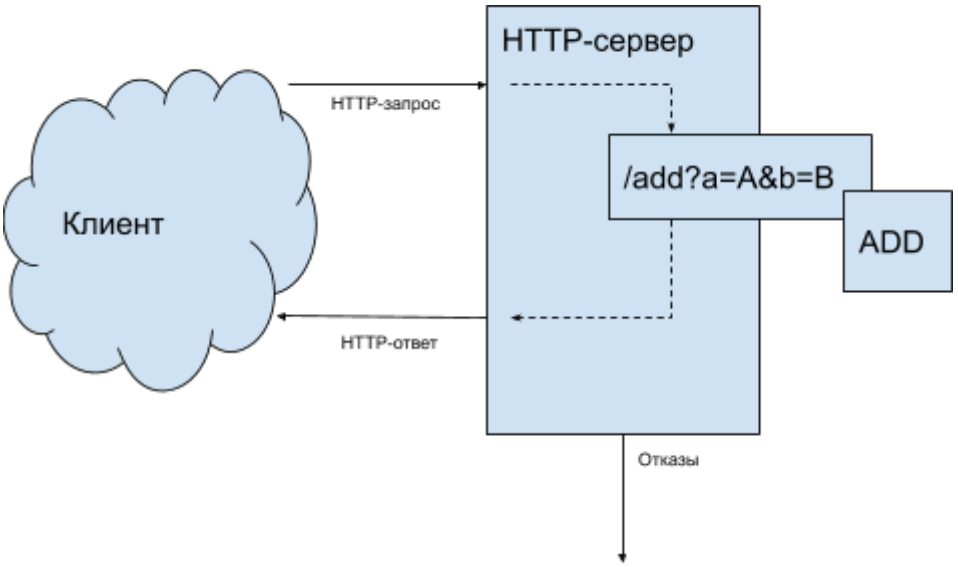


Рисунок 4

Функция сложения двух операндов `ADD` будет является процессом обслуживания. Очевидно, данный процесс характеризуется временем выполнения вычислений. Сервису нужно поместить операнды в память. Затем дождаться свободного кванта времени, чтобы выполнить команду ADD процессора. После считать полученный результат из результирующего регистра. Мы точно знаем сколько циклов процессора потребуется для выполнения инструкции сложения, но не знаем когда операционная система (ОС) отправит процесс на выполнение. Таким образом, время выполнения сложения - это случайная величина. Чтобы не отвлекаться на детали, здесь и далее мы будем говорить про абстрактную ОС, которая выполняет команду, если процессор свободен и отказывает в выполнении команды в противном случае.

Получив результат сложения, сервис формирует и отправляет HTTP-ответ, создавая выходной поток. Выходной поток также является случайным процессом, т.к. напрямую зависит от двух случайных процессов - входного потока и процесса обработки

Проведем анализ сервиса, представленного на рисунке 4. Предположим, он запущен в единственном экземпляре на однопроцессорной машине. Предположим, сервис получает постоянное число запросов каждый квант времени и на обработку каждого запроса тратит одно и то же время. Тогда СМО описывается формулой (3)

$$D_i/D_p/1 \text{ (3)}$$

$$D_p(t) = 0.5,\ \Delta t = 1ms \text{ (4)}$$

$$D_i(t) = 1,\ \Delta t = 1ms \text{ (5)}$$

Как указано формулами (4) и (5) сервис получает **один** запрос каждую миллисекунду и тратит **0.5ms** на его обработку. Временная диаграмма будет выглядеть следующим образом:

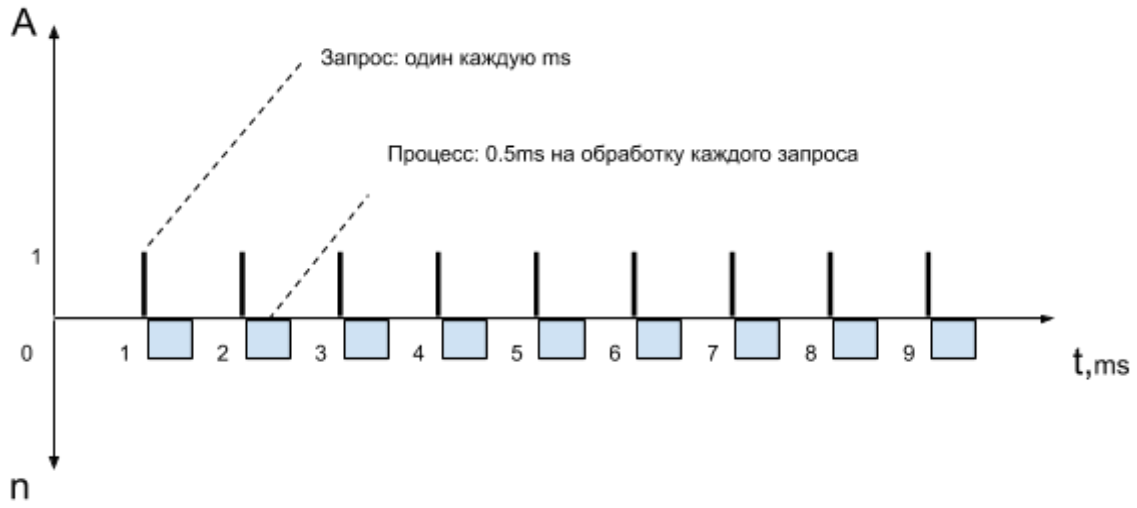


Рисунок 5

Как видно из рисунка 5, сервис будет обрабатывает 1000 запросов в секунду (**1Krps)** с задержкой **0.5ms.** При этом время процессора используется на **50%**. Сервис успешно справляется с нагрузкой. Если предположить, что процесс D_i представлен формулой (6), то временная диаграмма будет представлена на рисунке 6

$$D_i(t) = 2,\ \Delta t = 1ms \text{ (6)}$$

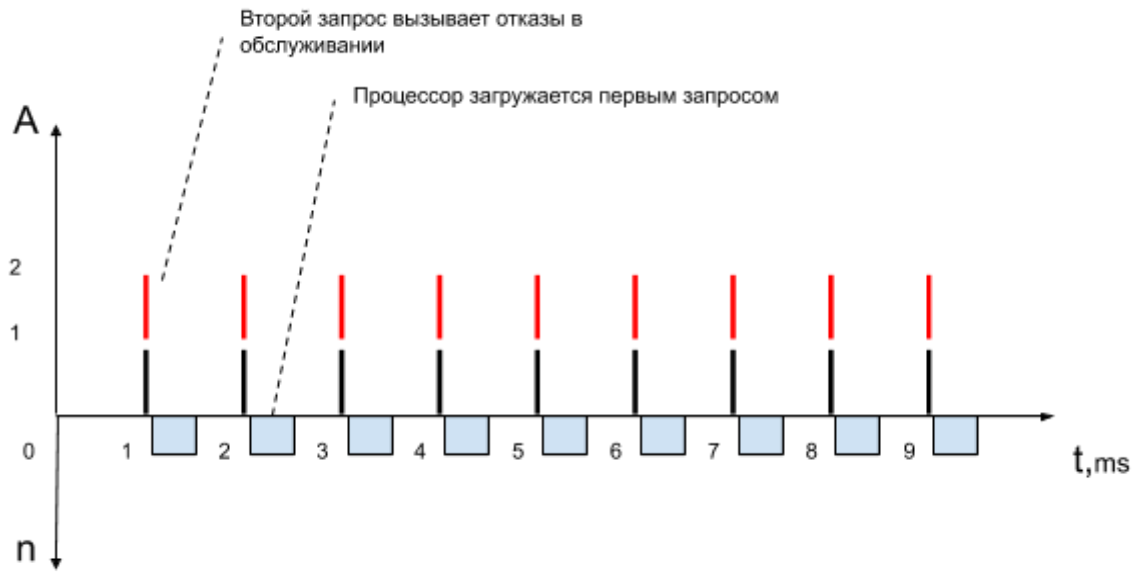


Рисунок 6

Каждую миллисекунду поступает **2 запроса**, но сервис всё равно обрабатывает **1Krps**, с задержкой в интервале **0.5ms**. Половина всех запросов вызывает отказ ОС, по причине загрузки процессора и формируют поток отказов, представленный формулой (7)

$$D_e(t) = 1,\ \Delta t = 1ms \quad (7)$$

Такой результат нас, конечно, не может устраивать. Однако, анализ СМО позволяет нам выдвигать различные гипотезы по входному потоку и процессу обслуживания. И в результате, понимать поведение сервиса под нагрузкой ещё до его реализации.

Решить проблему наличия отказов можно. Для этого необходимо изменить модель, которая описана формулой (3), с целью повышения ее производительности под входной поток, который описан формулой (6). Измененная модель описана формулой (8). Временная диаграмма представлена на рисунке 7

$$D_i/D_p/2 \quad (8)$$

Рисунок 7

По рисунку 7 видно, что производительность предложенной СМО составляет **2Krps** с задержкой **0.5ms**. Но мы вынуждены использовать дополнительный ресурс - ещё один процессор.

В рассмотренном примере чувствуется некоторая несправедливость. Модель формулы (3) теряет половину запросов, потребляя **50%** процессорного времени: **500ms** каждую **1s**. При этом модель формулы (8) работает без потерь, но каждый из двух процессоров также загружен лишь наполовину. Данную проблему решает СМО с ожиданием, которую принято описывать формулой (9)

$$A/B/n/q \text{ (9)}$$

, где A - входной поток, B - процесс обслуживания, n - число обработчиков, а q - предельный размер очереди сообщений. Если $q = \infty$, предполагается чистая теоретическая СМО с ожиданием. Если $q \in N,\ q > 0$, то предполагается СМО с ожиданием и ограничением.

Произведем замену модели формулы (3), не на модель формулы (8), а на модель формулы (10). Временная диаграмма такой СМО представлена на рисунке 8

$$D_i/D_p/1/1 \text{ (10)}$$

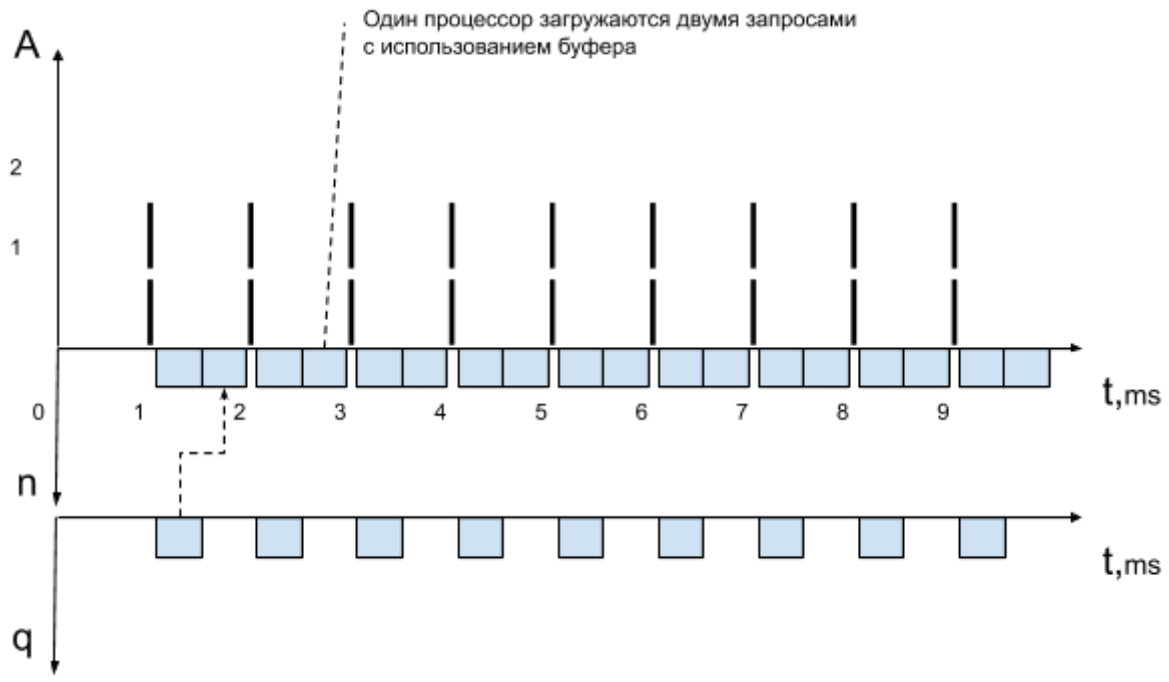


Рисунок 8

На рисунке 8 в начале каждой миллисекунды поступает два запроса. ОС отправляет первый запрос на выполнение в процессор, а второй запрос помещается в буфер. Как только процессор освобождается, ОС передает ему на выполнение запрос, находящийся в буфере. Таким образом, наш сервис будет обрабатывать **2Krps** сообщений с задержками **0.5ms** для нечетных запросов и **1ms** для четных запросов. СМО работает без потерь. При этом утилизация процессора будет составлять **100%**. Архитектура данного решения представлена на рисунке 9

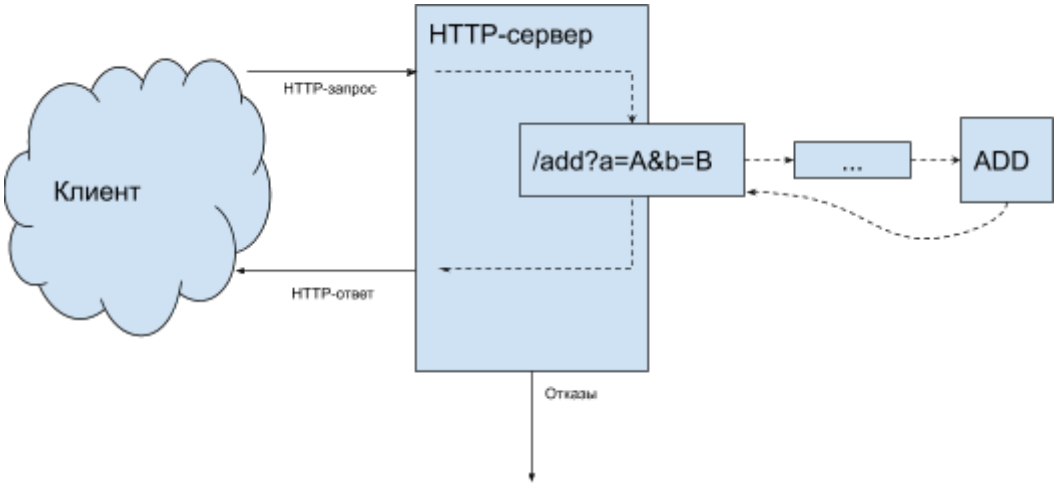


Рисунок 9

Согласно схеме на рисунке 9 HTTP-сервер получает запросы, которые помещает в очередь на выполнение. По мере освобождения обработчика сложения `ADD`, запросы считываются из очереди. Обработанный запрос с результатом возвращается обратно HTTP-серверу, который возвращает его

клиенту. В реальности сервис не будет получать сообщения с распределением постоянной величины. Время обработки сообщения также не будет постоянным. Такие процессы будут характеризоваться распределениями общего вида G. Огромный вклад в их изучение сделал советский академик Александр Яковлевич Хинчин, в 1932 году доказав формулу Поллачека-Хинчина.

Возвращаясь к определению, изложенному в разделе “Введение”, можно утверждать, что в данном примере СМО с ожиданием и ограничениями - это и есть *брокер сообщений*. При этом буфером будет являться очередь сообщений. В следующем разделе рассмотрим пример реализации брокера сообщений.

Реализация брокера сообщений

Итак, работа брокера сообщений может быть основана на использовании *очереди сообщений*. **Очередь сообщений** - это абстрактный компонент, управляющий областью памяти и обладающий дисциплиной доступа к данным, находящимся в данной памяти.

> **Важно!** Не путайте очередь как компонент брокера сообщений и очередь как абстрактный тип данных. Как абстрактный тип данных, очередь - это контейнер данных с дисциплиной доступа FIFO (first in first out). Стэк - контейнер данных с дисциплиной доступа LIFO (last in first out). Очередь как компонент брокера сообщений может использовать и как дисциплину FIFO, так и дисциплину LIFO, так и любую другую дисциплину: случайная выдача сообщений, агрегированная выдача, выдача сообщений по приоритетам и т.д.

Следует отметить, что брокеры сообщений не обязательно используют очереди сообщений как основной принцип функционирования. Брокер сообщений может представлять собой и СМО с отказами. На рисунке 10 представлена простейшая архитектура брокера сообщений, обеспечивающего передачу сообщений со всех источников всем приемникам:

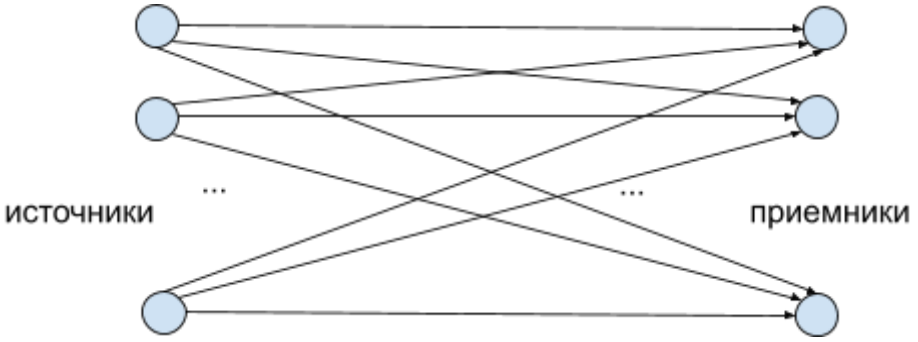


Рисунок 10

На рисунке 10 видно, что данная архитектура сложна обилием связей между источниками и приемникам. Каждый источник соединен с каждый приемником. Сложность данной архитектуры представляется как $O(n^2)$, но время доставки сообщения можно определить функцией Тау $T(1)$.

В начале прошлого века со столь высокой сложностью столкнулись и в области авиаперевозок. В итоге, в 1955 году компания Delta Air Lines разработала и внедрила архитектуру *spoke-hub*, основанную на элементах теории графов, в процесс организации авиаперелетов. Данная архитектура позволяет значительно уменьшить количество связей. Позже в 70-х годах подход стал использоваться в построении вычислительных сетей Ethernet, а сейчас используется для упрощения распределенных вычислительных систем - в частности, брокеров сообщений

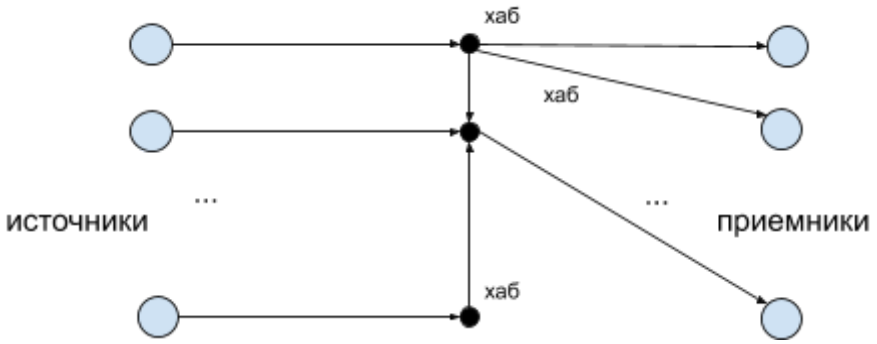


Рисунок 11

Данная архитектура предполагает, что каждый источник и приемник имеет связь лишь с собственным хабом. Источник передает сообщение хабу источника, хаб источника передает сообщение хабу приемника, хаб приемника передает сообщение приемнику. Как видно на рисунке 11, число присутствующих связей стало меньше, чем на схеме рисунка 10 и составляет $(n-1)$, где n - число источников, приемников и хабов. Таким образом сложность архитектуры составляет $O(n)$, а тау составляет $T(k)$, где k - число переходов сообщения от одного узла к другому по пути от источника к приемнику.

> *Для самостоятельного изучения*. На рисунке 11 приведена неоптимальная топология. Попробуйте максимально ее упростить. Оптимальное решение можно уточнить у преподавателя.

Обе эти архитектуры имеют достоинства и недостатки по критериям простоты и скорости доставки сообщений. Очереди сообщений являются компромиссным вариантом

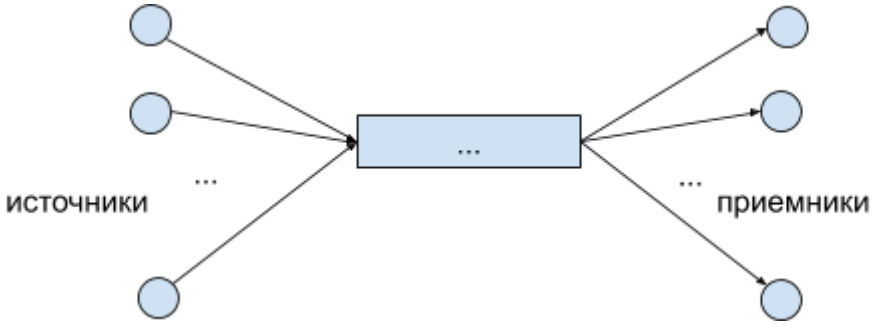


Рисунок 12

На схеме рисунка 12 можно отметить, что сложность связей данной архитектуры также составляет $O(n)$, где n - число источников и приемников. А функция Тау постоянна и равна $T(2)$, потому что сообщения переходят от источника к очереди, затем из очередь приёмнику. Реализуем небольшой модуль на языке Golang позволяющий работать с очередями сообщений.

Интерфейс данного модуля представлен ниже:

```
type Queue interface {
  Pop() (string, error)
  Push(string) error
}
```

Напишем имплементацию данного интерфейса с использованием базы данных (БД) redis. Для доступа к БД будем использовать рекомендуемую создателями БД библиотеку radix.

```
var (
  connFunc = func(network, addr string) (radix.Conn, error) {
     return radix.Dial(network, addr,
        radix.DialTimeout(timeout),
     )
  }
)

func NewQueue(name, address string) Queue {
  pool, err := radix.NewPool("tcp", address, 1,
radix.PoolConnFunc(connFunc))
  if err != nil {
     panic(err)
  }
  return &RedisQueue{
     addr: address,
     name: name,
     pool: pool,
  }
}

func (q *RedisQueue) Pop() (string, error) {
  var result string
  err := q.pool.Do(radix.Cmd(&result, "LPOP", q.name))
  return result, err
}

func (q *RedisQueue) Push(v string) error {
  return q.pool.Do(radix.Cmd(nil, "RPUSH", q.name, v))
}
```

Для самостоятельного изучения. Как вы заметили, данный код основан на типе данных List БД redis. В примере выше реализована очередь сообщений с дисциплиной доступа FIFO. Модифицируйте данный модуль и реализуйте дисциплину доступа LIFO. Правильный ответ вы можете обсудить с преподавателем.

Таким образом, мы реализовали простейший брокер сообщений, использующий очередь сообщений. Протестировать его можно следующим кодом:

```
ss := []string{"1", "2", "3", "4", "5", "6", "7", "8", "9"}

q := NewQueue("test_queeu", "127.0.0.1:6379")
for _, s := range ss {
  err := q.Push(s)
  c.Assert(err, IsNil)
}

for i := 0; i < len(ss); i += 1 {
  s, err := q.Pop()
  c.Assert(err, IsNil)
  c.Assert(s, Equals, ss[i])
}
```

Для самостоятельного изучения. Подумайте, какие типы данных БД redis можно использовать для поддержки таких дисциплин доступа, как случайная выдача сообщений и выдача сообщений по приоритету? Ответ можете уточнить у преподавателя.

Очень часто взаимодействие с брокерами сообщений происходит с использованием паттерна Publish-Subscribe. Паттерн был впервые предложен профессором Университета Корнелл Кеном Бирманном в 1987 году на конференции SOSP-87. Диаграмма последовательности представлена на рисунке 13

Рисунок 13

На диаграмме рисунка 13 два приёмника регистрируются в брокере для получения сообщений. В момент, когда источник публикует сообщение, оба приёмника получают его. После чего приёмник

подтверждает корректное получение сообщения. Процесс подтверждения получения применяется не всегда, а только для дисциплин *at least once* и *exactly once*. Реализация подобного брокера сообщений с использованием БД redis представлена ниже:

```
type (
  PubSub interface {
     Publish(string) error
     Subscribe(chan<- radix.PubSubMessage) error
     Unsubscribe(chan<- radix.PubSubMessage) error
  }

  RedisPubsub struct {
     name string
     pool *radix.Pool
     pubsub radix.PubSubConn
  }
)

var (
  connFunc = func(network, addr string) (radix.Conn, error) {
     return radix.Dial(network, addr,
        radix.DialTimeout(timeout),
     )
  }
)

func NewPubSub(name, address string) PubSub {
  pool, err := radix.NewPool("tcp", address, 1,
radix.PoolConnFunc(connFunc))
  if err != nil {
     panic(err)
  }
  pubsub, err := radix.PersistentPubSubWithOpts("tcp", address)
  if err != nil {
     panic(err)
  }
  return &RedisPubsub{
     name: name,
     pool: pool,
     pubsub: pubsub,
  }
}

func (ps *RedisPubsub) Publish(v string) error {
  return ps.pool.Do(radix.Cmd(nil, "PUBLISH", ps.name, v))
}

func (ps *RedisPubsub) Subscribe(ch chan<- radix.PubSubMessage) error {
  return ps.pubsub.Subscribe(ch, ps.name)
}

func (ps *RedisPubsub) Unsubscribe(ch chan<- radix.PubSubMessage) error {
  return ps.pubsub.Unsubscribe(ch, ps.name)
}
```

Протестировать полученное решение можно следующим образом:

```
const alphabet = "abcdefghijklmnopqrstuvwxyz"
```

```
func (s *PubSubSuite) TestPubSub(c *C) {
  var (
     ch1 = make(chan radix.PubSubMessage)
     ch2 = make(chan radix.PubSubMessage)

     err error
     wg = &sync.WaitGroup{}
  )

  p := NewPubSub("topic", "127.0.0.1:6379")
  s1 := NewPubSub("topic", "127.0.0.1:6379")
  s2 := NewPubSub("topic", "127.0.0.1:6379")

  err = s1.Subscribe(ch1)
  c.Assert(err, IsNil)
  err = s2.Subscribe(ch2)
  c.Assert(err, IsNil)

  var  (
     sb1 = strings.Builder{}
     sb2 = strings.Builder{}
  )
  wg.Add(2)
  go func() {
     for v := range ch1 {
        sb1.Write(v.Message)
     }
     wg.Done()
  }()
  go func() {
     for v := range ch2 {
        sb2.Write(v.Message)
     }
     wg.Done()
  }()

  for _, v := range alphabet {
     err := p.Publish(string(v))
     c.Assert(err, IsNil)
  }

  <-time.NewTicker(time.Second).C
  err = s1.Unsubscribe(ch1)
  c.Assert(err, IsNil)
  close(ch1)
  err = s2.Unsubscribe(ch2)
  c.Assert(err, IsNil)
  close(ch2)

  wg.Wait()
  c.Assert(sb1.String(), Equals, alphabet)
  c.Assert(sb2.String(), Equals, alphabet)
}
```

Сравнение существующих брокеров сообщений

В настоящее время нет необходимости реализовывать брокеры сообщений самостоятельно, так как в мире представлено множество готовых брокеров сообщений. Существуют проприетарные решения: IBM MQ, Google Cloud Pub/Sub, Amazon SQS и т.д. Существуют и разработанные сообществом брокеры с открытым исходным кодом: Apache Kafka, RabbitMQ, NATS, NSQ и т.д.

Ниже представлена сравнительная таблица популярных брокеров сообщений по их основным характеристикам:

Таблица 1

	Apache Kafka	RabbitMQ	NATS
Язык разработки	Scala	Erlang	Golang
Год первого релиза	2011	2007	2015
Клиенты	17 языков	30 языков	12 языков
Паттерн интерфейса	Message queue Publish-Subscribe	Message queue Publish-Subscribe	Message queue Publish-Subscribe Request-reply *
Протоколы	TCP	AMQP, STOMP, MQTT	Google Protobuf
Дисциплина доставки сообщений	at least once at most once exactly once	at least once at most once	at least once at most once
Тип хранилища	Диск	Память Диск	Память Диск

Отдельного вопроса заслуживает пропускная способность брокеров сообщений. На рисунках 14-16 приведены некоторые результаты нагрузочного тестирования рассматриваемых брокеров сообщений

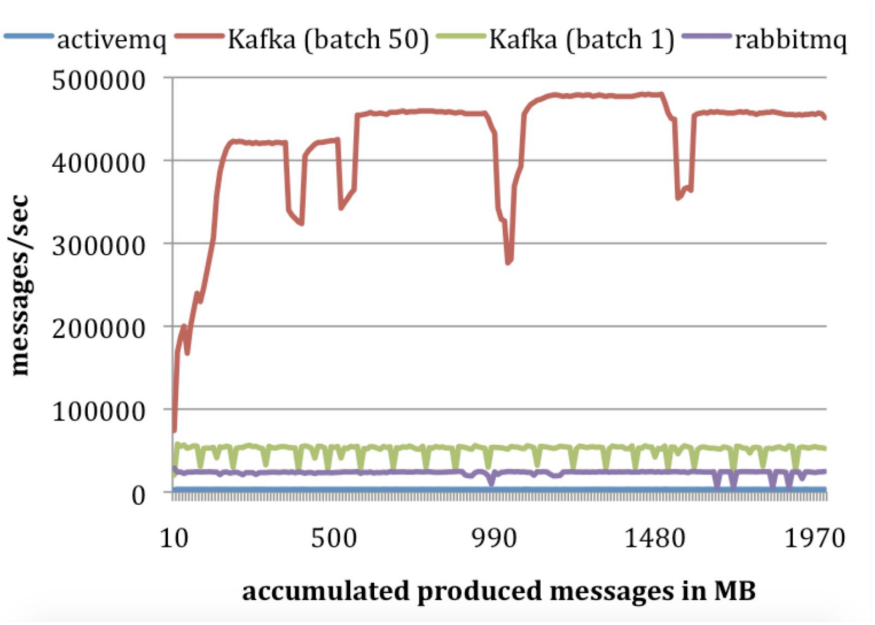


Рисунок 14. Сравнение пропускной способности Apache Kafka и RabbitMQ. Источник

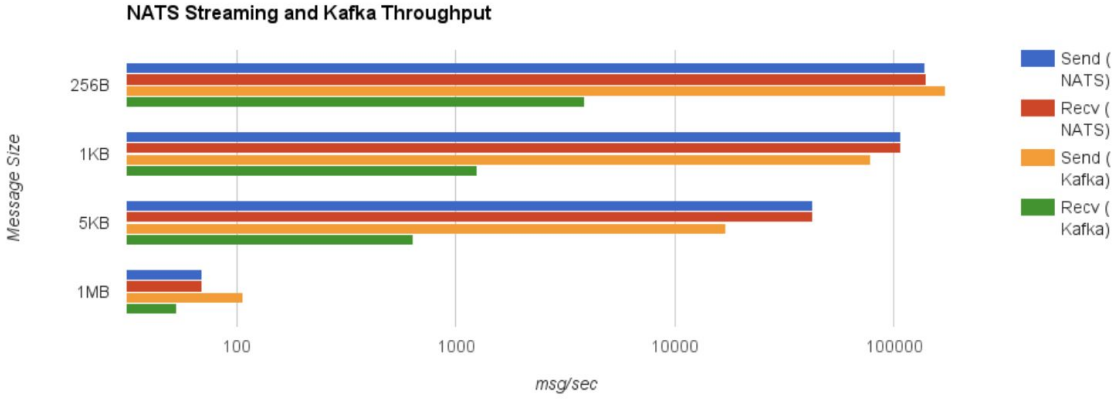
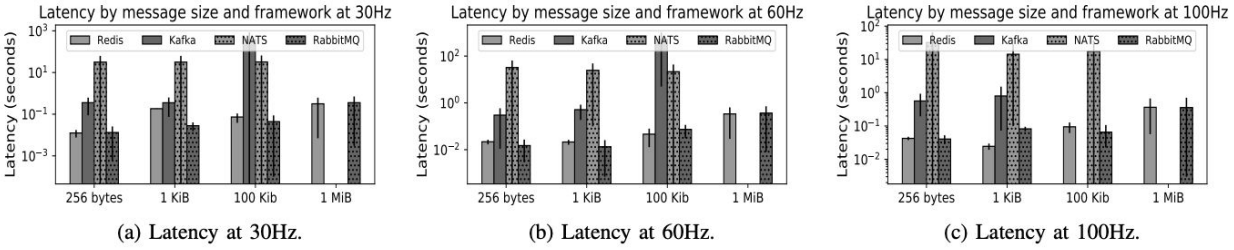


Рисунок 15. Сравнение пропускной способности Apache Kafka и NATS. Источник



(a) Latency at 30Hz. (b) Latency at 60Hz. (c) Latency at 100Hz.

Рисунок 16. Задержки доставки сообщений: Apache Kafka, RabbitMQ, NATS. Источник

Следует отметить, что результаты нагрузочного тестирования зависят от методологии. Но в целом, по указанным выше цифрам можно сделать вывод, что Apache Kafka является лучшим выбором для обработки данных в реальном времени. Не стоит обделять вниманием и активно развивающийся на данный момент NATS, написанный на Golang и показывающий сравнимую производительность с

Apache Kafka. Однако, следует уделить внимание задержкам в доставке ёмких сообщений. В случае необходимости интеграции со сторонними решениями может подойти RabbitMQ. Он является лидером по количеству поддерживаемых языков программирования и протоколам взаимодействия

Разработка сервиса оценок

В предыдущем разделе мы получили сравнительный анализ различных брокеров сообщений, Теперь разработаем сервис сбора оценок пользователей, аналогичный emojicom. Сервис представляет собой HTTP-сервер с двумя обработчиками:

- Получение оценки пользователя `rate`
- Возврат результирующей оценки `total`

Обработчик `rate` получает оценку пользователя параметром запроса. Оценка должна учитываться с использованием скользящего среднего (11) и сохраняться в БД результата. Обработчик `total` получает накопленную оценку из БД и возвращает ее клиенту. Архитектура решения представлена на рисунке 8

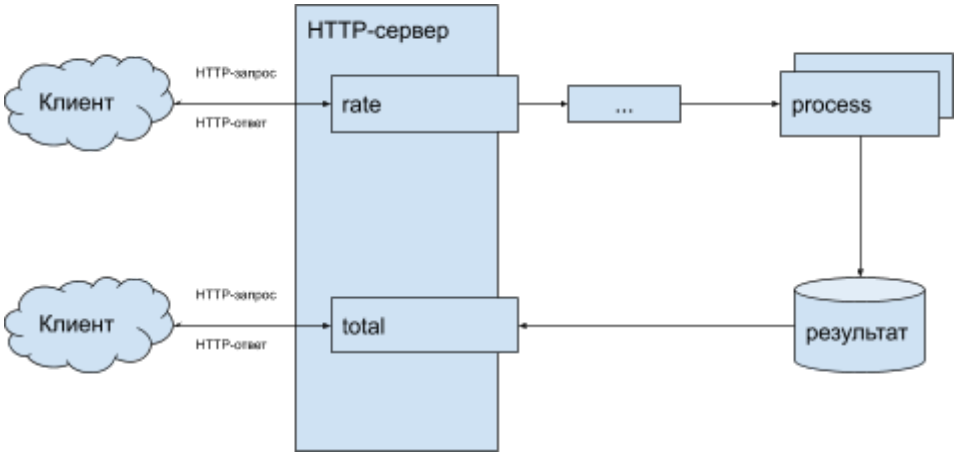


Рисунок 8

Как показано на рисунке 8 оценки от пользователей предварительно попадают в бокер сообщений. Из брокера сообщений их считывают обработчики `process`, непосредственно выполняющие сохранение оценок для последующего вычисления результата. В качестве брокера сообщений будем использовать Apache Kafka.

$$\frac{1}{n}\sum_{0}^{n-1} p_{t-i} \quad (11)$$

Параметр n формулы (11) определяет размер окна, по которому вычисляется скользящее среднее оценки. Если мы используем redis в качестве хранилища оценок, псевдокод обработчика `process` будет следующим:

```
LPUSH rate value
```

```
LTRIM rate n
```

Вызов команды `LTRIM` предназначен для устойчивого потребления памяти базой данных. Для возврата результата обработчику `total` нужно достать предыдущие n оценок, используя команду `LRANGE`, вычислить среднее значение по формуле (11) и вернуть его клиенту

Опишем инфраструктуру нашего решения. Воспользуемся готовым образом Apache Kafka от компании Bitnami, которая любезно его подготовила, вынеся необходимые настройки в переменные окружения.

```
version: "3.7"
services:

 api:
   build:
     dockerfile: Dockerfile
     context: .
   command: "./bin/api"
   environment:
     - KAFKA_BROKERS=kafka:9092
   ports:
     - 8081:80
   depends_on:
     - kafka
     - redis

 process:
   build:
     dockerfile: Dockerfile
     context: .
   command: "./bin/process"
   environment:
     - KAFKA_BROKERS=kafka:9092
   depends_on:
     - kafka
     - redis

 kafka:
   image: docker.io/bitnami/kafka:2-debian-10
   ports:
     - 9092:9092
   environment:
     - KAFKA_BROKER_ID=1
     - KAFKA_LISTENERS=PLAINTEXT://:9092
     - KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092
     - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
     - ALLOW_PLAINTEXT_LISTENER=yes
   depends_on:
     - zookeeper
   healthcheck:
     test:
       ["CMD", "kafka-topics.sh", "--list", "--zookeeper",
"zookeeper:2181"]
     interval: 30s
     timeout: 10s
     retries: 4
```

```
  redis:
    image: redis:6.0
    ports:
      - 6379:6379

  zookeeper:
    image: docker.io/bitnami/zookeeper:3-debian-10
    ports:
      - 2181:2181
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
```

Так как старт Apache Kafka занимает некоторое время, проверить его готовность к обработке запросов можно использованием определенного `healthcheck`:

```
d2fb31517f0f   bitnami/kafka:latest       "/opt/bitnami/script…"   3
minutes ago   Up 3 minutes (healthy)   0.0.0.0:9092->9092/tcp
geekbrainsqueues_kafka_1
```

Теперь проверим, что Apache Kafka работает корректно и приёмник получает сообщения от источника. Для этого воспользуется утилитой gocdk-pubsub и запустим приёмник:

```
KAFKA_BROKERS=127.0.0.1:9092 gocdk-pubsub sub -n 0
'kafka://group?topic=test'
Receiving messages from "kafka://group?topic=test"...
```

Теперь отправим несколько сообщений со стороны источника:

```
$: echo '1' | KAFKA_BROKERS=127.0.0.1:9092 gocdk-pubsub pub 'kafka://test'
Enter messages, one per line, to be published to "kafka://test".
$: echo '2' | KAFKA_BROKERS=127.0.0.1:9092 gocdk-pubsub pub 'kafka://test'
Enter messages, one per line, to be published to "kafka://test".
```

Удостоверимся, что приёмник получает сообщения, отправленные источником:

```
KAFKA_BROKERS=127.0.0.1:9092 gocdk-pubsub sub -n 0
'kafka://group?topic=test'
Receiving messages from "kafka://group?topic=test"...
1
2
```

Опишем обработчики `rate` и `total`. Для взаимодействия с Apache Kafka будем использовать универсальную библиотеку Cloud Development Kit, разработанную компанией Google. Библиотека содержит абстракцию Pub/Sub для доступа к различным БС. На данный момент поддерживаются как Apache Kafka, RabbitMQ и NATS, рассмотренные нами в разделе “Сравнение существующих брокеров сообщений”, так и проприетарные Google Cloud Pub/Sub, Amazon SNS, Amazon SQS, Azure Service Bus. Ниже приведен исходный код обработчиков:

```
func init() {
  router.
     HandleFunc("/rate", PostRateHandler).
     Methods(http.MethodPost)
```

```
    router.
        HandleFunc("/total", GetTotalHandler).
        Methods(http.MethodGet)
}

func main() {
    if err := web.ListenAndServe(); err != http.ErrServerClosed {
        panic(fmt.Errorf("error on listen and serve: %v", err))
    }
}

func GetTotalHandler(w http.ResponseWriter, r *http.Request) {
    var rates []string
    err := storage().Do(radix.Cmd(&rates, "LRANGE", "result", "0", "10"))
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
    if len(rates) == 0 {
        w.WriteHeader(http.StatusServiceUnavailable)
        return
    }

    var sum int
    for _, rate := range rates {
        v, err := strconv.Atoi(rate)
        if err != nil {
            continue
        }
        sum += v
    }
    result := float64(sum)/float64(len(rates))
    _, _ = w.Write([]byte(fmt.Sprintf("%.2f", result)))
}

func PostRateHandler(w http.ResponseWriter, r *http.Request) {
    rate := r.FormValue("rate")
    if _, err := strconv.Atoi(rate); err != nil {
        w.WriteHeader(http.StatusBadRequest)
        return
    }
    err := topic().Send(context.Background(), &pubsub.Message{
        Body: []byte(rate),
    })
    if err != nil {
        log.Println(err)
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
}
```

Инициализация клиентов для доступа к брокеру сообщений и БД результата представлена ниже:

```
func topic() *pubsub.Topic {
    if t != nil {
        return t
    }
    var err error
    t, err = pubsub.OpenTopic(context.Background(), "kafka://rates")
    if err != nil {
```

```
        panic(err)
    }
    return t
}

func storage() *radix.Pool {
    if s != nil {
        return s
    }
    var err error
    s, err = radix.NewPool("tcp", "redis:6379", 1,
radix.PoolConnFunc(connFunc))
    if err != nil {
        panic(err)
    }
    return s
}
```

Исходный код обработчика `process` приведен ниже:

```
func subscription() (*pubsub.Subscription, error) {
    if sub != nil {
        return sub, nil
    }
    var err error
    sub, err = pubsub.OpenSubscription(context.Background(),
"kafka://process?topic=rates")
    if err != nil {
        return nil, err
    }
    return sub, nil
}

func main() {
    for {
        s, err := subscription()
        if err != nil {
            log.Println(err)
            time.Sleep(time.Second)
            continue
        }
        msg, err := s.Receive(context.Background())
        if err != nil {
            log.Println(err)
            time.Sleep(time.Second)
            continue
        }

        err = storage().Do(radix.Cmd(nil, "LPUSH", "result",
string(msg.Body)))
        if err != nil {
            log.Println(err)
        }
        if rand.Float64() < .05 {
            _ = storage().Do(radix.Cmd(nil, "LTRIM", "result", "0", "9"))
        }
        msg.Ack()
    }
}
```

Теперь выполним запись нескольких оценок использованием с утилиты wrk. Для этого создадим небольшой тест, отправляющий случайную оценку от 0 до 10:

```
math.randomseed(os.time())
request = function()
  local k = math.random(0, 10)
  local url = "/rate?rate="..k
  return wrk.format("POST", url)
end
```

Запустим его из командной строки:

```
$: wrk -c5 -t5 -d1m -s ./wrk.lua 'http://127.0.0.1:8081'
Running 1m test @ http://127.0.0.1:8081
  5 threads and 5 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    30.15ms   34.23ms 379.98ms   93.32%
    Req/Sec    42.87     17.94   110.00     73.69%
  12428 requests in 1.00m, 0.89MB read
Requests/sec:    206.89
Transfer/sec:     15.15KB
```

Итак, наш сервис обрабатывает **~200rps** со средней задержкой в **30ms**. Давайте проверим состояние окна для вычисления средней оценки в БД результата:

```
$: docker exec -it geekbrainsqueues_redis_1 redis-cli llen result
(integer) 26
```

Несмотря на то, что мы отправили больше **12428** запросов, размер окна составляет всего 26 элементов, что укладывается в вероятность **5%**, заложенную нами для его коррекции. Проверим среднюю оценку, полученную в результате теста:

```
$: curl 'http://127.0.0.1:8081/total'
4.55
```

Таким образом, мы реализовали сервис сбора оценок с использованием архитектурного паттерна брокер сообщений. Такое решение позволяет нам осуществлять масштабирование сервиса, сохраняя его надежность. Мы можем запустить несколько HTTP-серверов, которые будут заполнять очередь сообщений нашего брокера, сохраняя низкие параметры задержки для клиентов. Мы можем запустить и несколько обработчиков `process`, если единственный не будет справляться, используя дисциплину доставки *exactly once*. При этом, в случае отказа HTTP-сервера и/или обработчика `process`, сообщения не теряются, а сохраняются в очереди брокера

Практические задания

1. Запустите большее число копий HTTP-сервера. Эмулируйте нагрузку как она может поступать от балансировщика равномерно на все копии сервиса. Оцените полученные результаты: как меняется **rps** нашего сервиса? Как изменяется величина задержки?
2. Самостоятельно замените брокер сообщений, используя образ RabbitMQ или образ NATS. Оцените полученные результаты: как меняется **rps** нашего сервиса? Как изменяется величина задержки?
3. Модифицируйте HTTP-сервис таким образом, чтобы можно было запустить несколько обработчиков `process`, каждый из которых будет обрабатывать своё сообщение
4. Повторите п.1, но в течение теста погасите один из обработчиков, созданных в п.3. Убедитесь, что все запросы были обработаны корректно

Используемые источники

1. Книга Web Scalability for Startup Engineers о том, как разрабатывать масштабируемые сервисы
2. Статья А. К. Эрланга, с которой началась Теория массового обслуживания: The Theory of Probabilities and Telephone Conversations
3. Список публикаций А. Н. Колмогорова
4. Библиография А. Я. Хинчина
5. Статья «О большое и о малое»
6. Краткий список типов данных БД redis
7. Документация по Apache Kafka
8. Документация RabbitMQ
9. Документация NATS
10. Универсальная библиотека для работы с брокерами сообщений от компании Google