DINS®

Реализуем распределённый Rate-Limiter на Apache Ignite

О докладчике:

- Бухтояров Владимир. Опыт коммерческой разработки на Java 12 лет. Из них 7 лет в компании DINS.
- В настоящее время работаю в должности Java Lead Developer.
- Люблю заниматься параллельными алгоритмами и Grid Computing.
- Активно участвую в OpenSource.

Связь докладчика и темы:

Автор библиотеки Bucket4j - самой популярной реализации алгоритма Token Bucket для Java:

- 100 тысяч скачиваний с Maven Central ежемесячно.
- В зависимостях у 2500 проектов по данным github.
- Используется в JHipster, Jira, Armeria, Jobby, Kubernatess Java Client.

План доклада:

- Общий обзор задач ограничения пропускной способности.
- Специфика троттлинга в распределенной среде.
- Обзор алгоритма Token Bucket.
- Реализуем распределенный Token Bucket с использованием технологии Apache Ignite.
- Устраним типичные ошибки производительности в первоначального решения.

Когда возникает необходимость ограничивать пропускную способность:

- Защита нашей системы от нагрузки, порождаемой внешними запросами.
- Реализация контрактных обязательств.
- Fraud/anomaly Detection.
- Защита сторонних систем от нагрузки, которую порождает наша система.

Защищаем BOT API корпоративного мессенджера

```
@RestController
@AllArgsConstructor
public class MessageEndpoint {

    private final MessageService messageService;

    @RequestMapping(method = POST, value = "/message")
    public void sendMessage(Message message) {
        messageService.sendMessage(message);
    }

}
```

Где разместим троттлинг?

Защищаем BOT API корпоративного мессенджера

```
@RestController
@AllArgsConstructor
public class MessageEndpoint {

    private final MessageService messageService;

    @RequestMapping(method = POST, value = "/message")
    public void sendMessage(Message message) {
        int requiredPermits = message.estimateCost();
        RateLimiter rateLimiter = SecurityContext.getRateLimiter();
        if (!rateLimiter.tryAcquire(requiredPermits)) {
            throw new RateLimitException();
        }
        messageService.sendMessage(message);
    }

}
```

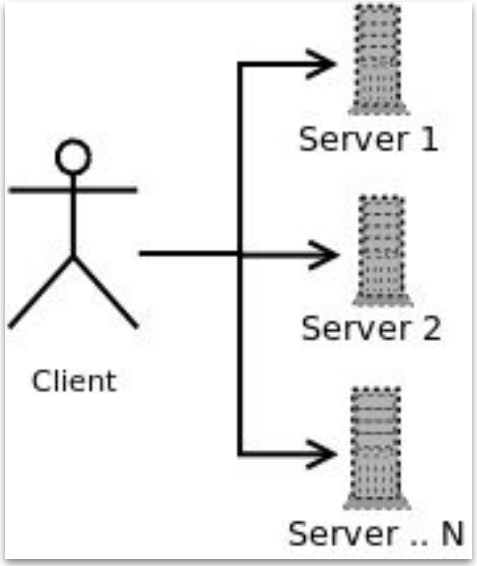
Минимально необходимый интерфейс:

```
public interface RateLimiter {

    boolean tryAcquire(int permits);

}
```

RateLimiter в кластере, почему плохо работают простые подходы без репликации состояния?



Нельзя просто взять лимит X/T и преобразовать в X/N/T:

- Underconsumption при неравномерном распределении запросов.
- Необходимость переконфигурирования всех серверов при смене топологии

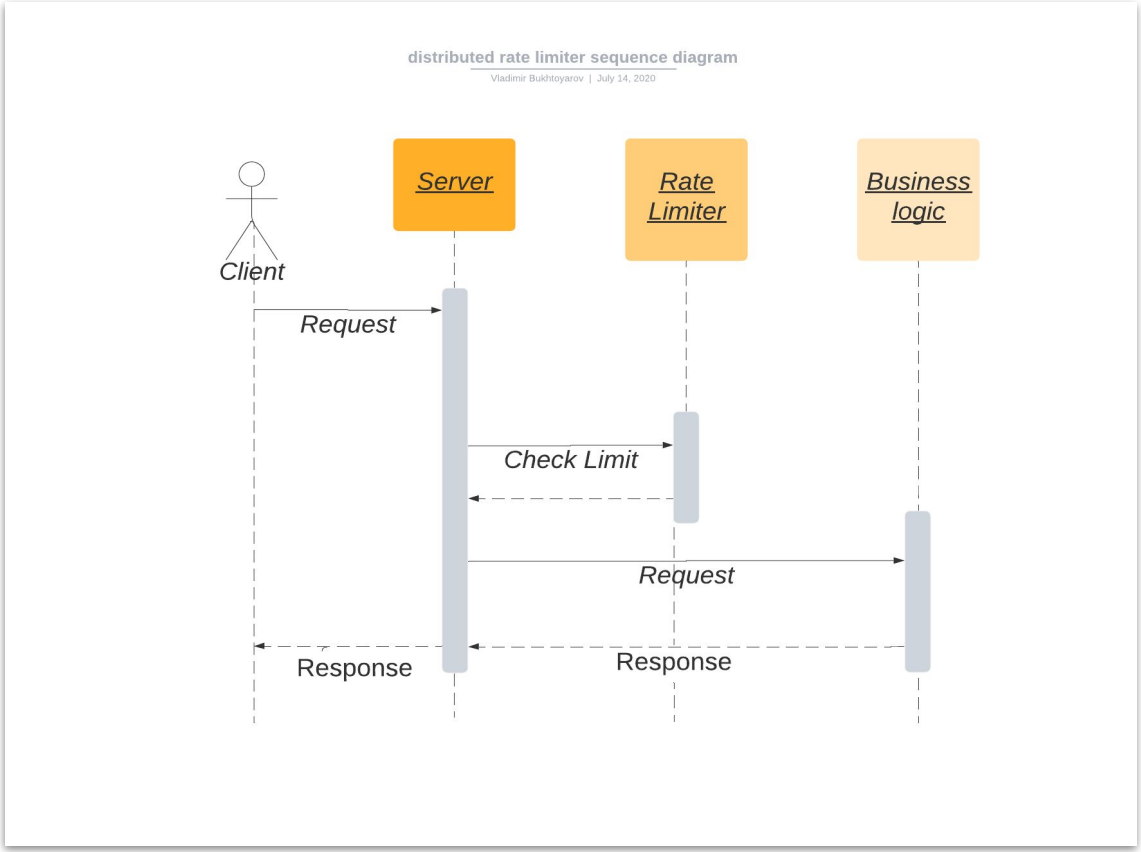
Классификация подходов репликации состоянии для распределенного RateLimiter:

- Централизованный синхронный:
  https://github.com/mokies/ratelimitj
  https://github.com/vladimir-bukhtoyarov/bucket4j
  https://resilience4j.readme.io/docs/ratelimiter

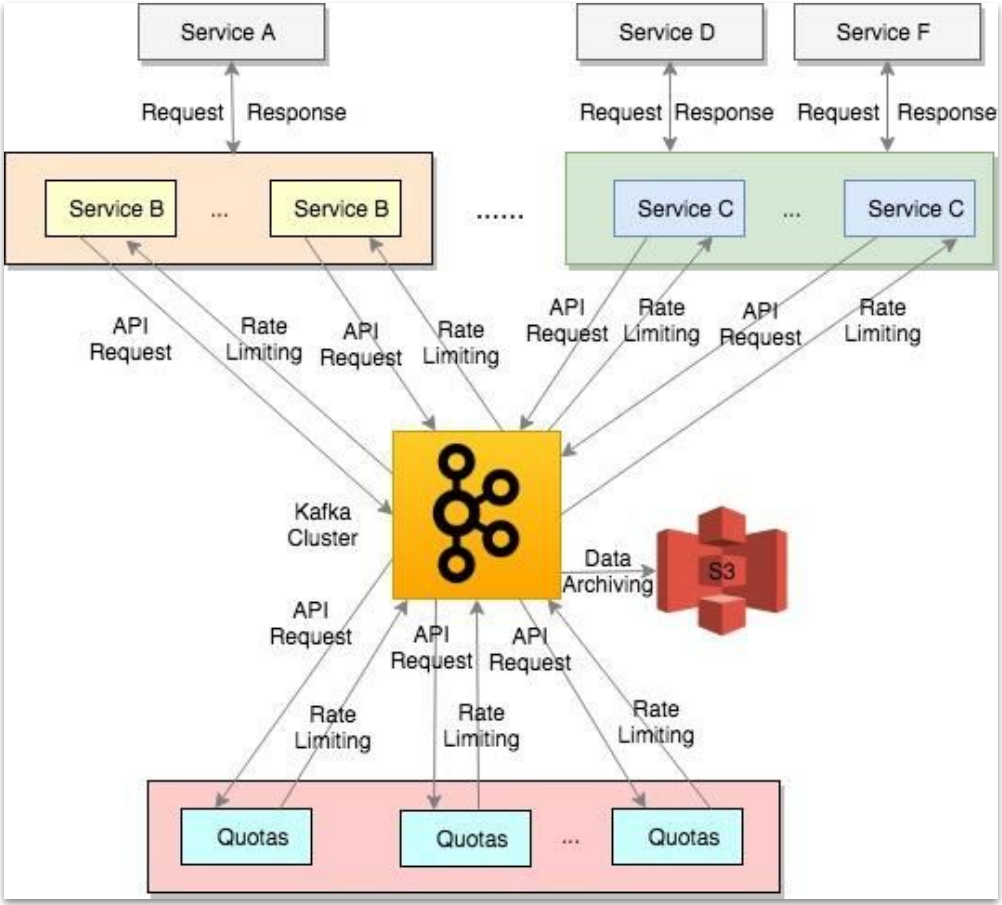
- Централизованный асинхронный:
  https://engineering.grab.com/quotas-service
  https://www.mailgun.com/blog/gubernator-cloud-native-distributed-rate-limiting-microservices/

- Децентрализованный:
  https://www.microsoft.com/en-us/research/wp-content/uploads/2007/01/fp076-raghavan.pdf

Централизованный синхронный RateLimiter



Централизованный асинхронный RateLimiter



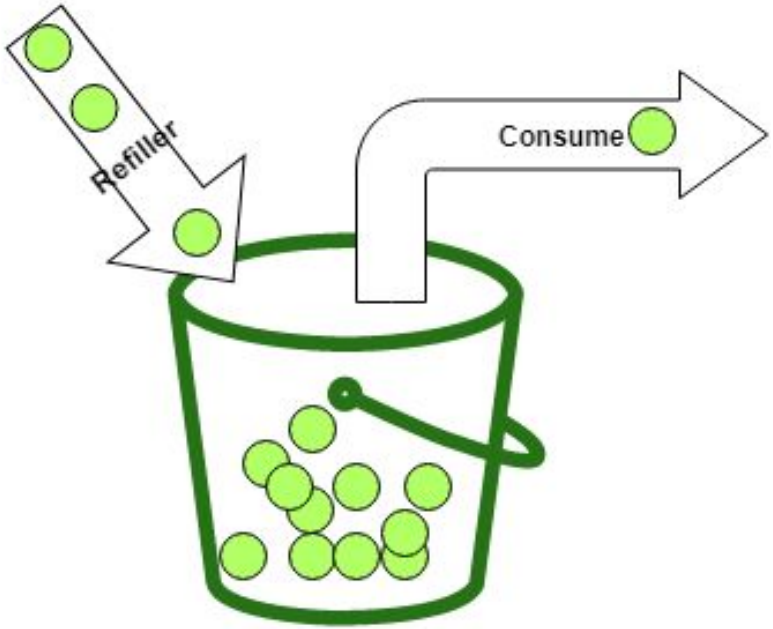
https://engineering.grab.com/quotas-service

DINS

Выбор алгоритма для Distributed RateLimiter

- Обзор алгоритма TokenBucket.
- Пример работоспособной реализации для одной JVM.

Token Bucket: алгоритм с фиксированной памятью



Потребляемая память:

Объем ведра	8 байт
Текущее число токенов в ведре	8 байт
Число наносекунд на генерацию одного токена	8 байт
Заголовок объекта	16 байт
Итого:	40 байт

https://golb.hplar.ch/2019/08/rate-limit-bucket4j.html

DINS

Token Bucket: алгоритм с фиксированной памятью

The token bucket algorithm can be conceptually understood as follows:

- A token is added to the bucket every `1/r` seconds.
- The bucket can hold at the most M If a token arrives when the bucket is full, it is discarded.
- When a packet (network layer PDU) of *n* bytes arrives:
  - if at least *n* tokens are in the bucket, *n* tokens are removed from the bucket, and the packet is sent to the network.
  - if fewer than *n* tokens are available, no tokens are removed from the bucket, and the packet is considered to be *non-conformant*.

https://en.wikipedia.org/wiki/Token_bucket

TokenBucket структура данных:

```
public class SchedulerFreeTokenBucket {

    private final long capacity;
    private long availableTokens;

    private final long nanosToGenerateToken;
    private long lastRefillNanotime;

    public SchedulerFreeTokenBucket(long permits, Duration period) {
        this.nanosToGenerateToken = period.toNanos() / permits;
        this.lastRefillNanotime = System.nanoTime();

        this.capacity = permits;
        this.availableTokens = permits;
    }
```

TokenBucket алгоритм:

```java
@Override
synchronized public boolean tryAcquire(int permits) {
    refill();
    if (availableTokens < permits) {
        return false;
    } else {
        availableTokens -= permits;
        return true;
    }
}

private void refill() {
    long now = System.nanoTime();
    long nanosSinceLastRefill = now - this.lastRefillNanotime;
    if (nanosSinceLastRefill <= nanosToGenerateToken) {
        return;
    }
    long tokensSinceLastRefill = nanosSinceLastRefill / nanosToGenerateToken;
    availableTokens = Math.min(capacity, availableTokens + tokensSinceLastRefill);
    lastRefillNanotime += tokensSinceLastRefill * nanosToGenerateToken;
}
```

Distributed TokenBucket выбор технологии для реализации:

- Обзор требований.
- Муки выбора.

Наши требования к RateLimiter в кластере:

- Работать быстро. Желательно укладываться в 1-2 миллисекунды.
- Корректно работать в условиях параллельного доступа.
- Выход из строя нескольких узлов не должно являться проблемой.
- Решение должно основываться на известных проверенных инструментах.

Чего не требуется:

- Хранение данных на дисках.
- Поддержка множества языков программирования, нам достаточно JVM based.

Пути реализации RateLimiter в кластере:

- DBMS + Select for Update?
  Всегда 3 сетевых запроса. Сложность настройки репликации.

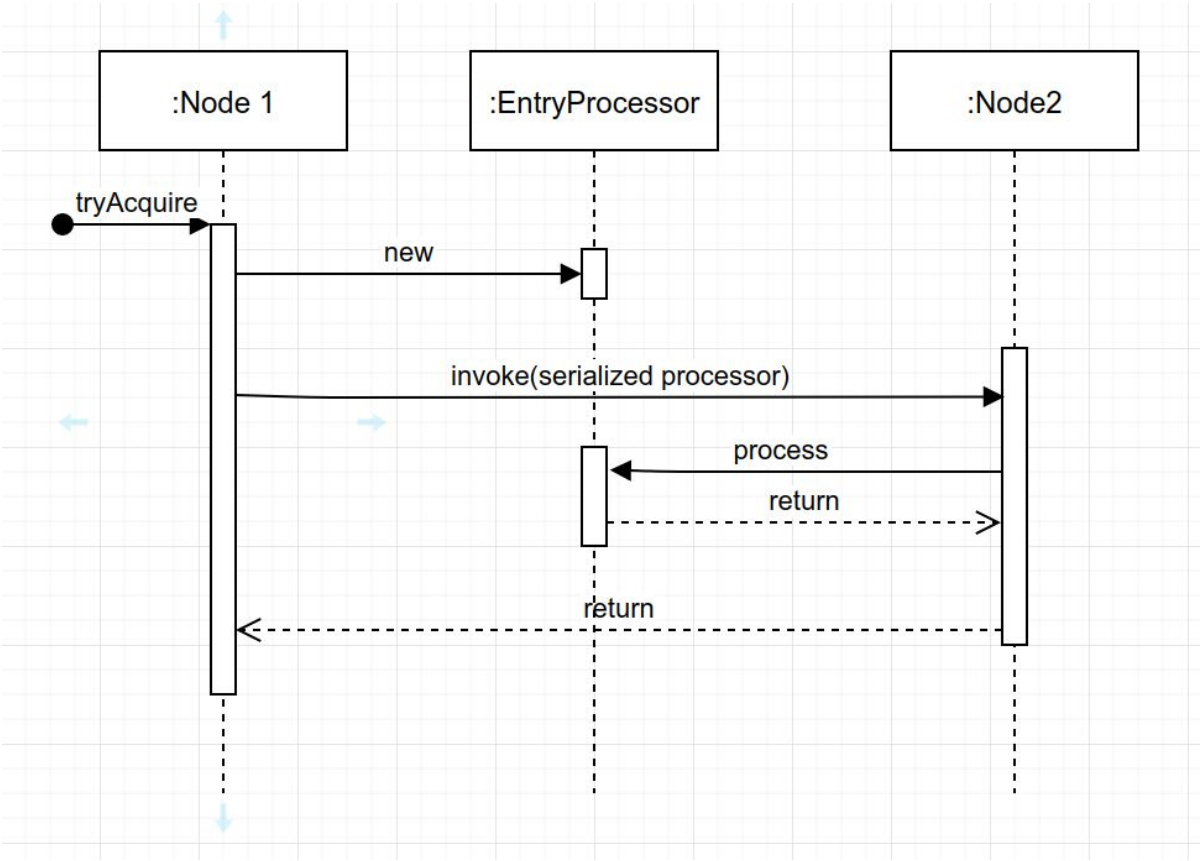
- MemCached, MongoDB,... + Compare And SWAP?!
  Чаще всего 2 запроса, однако в случае высокой конкуренции число CAS циклов не известно.

- DBMS + Stored procedures?!
  Всегда 1 сетевой запрос. Сложность настройки репликации.

- Redis, Tarantool, Aerospike + Stored procedures!
  Всегда 1 сетевой запрос.

- JSR 107(Hazelcast, Coherence, Ignite) + EntryProcessor!!!
  Всегда 1 сетевой запрос. Не требует дополнительных серверов!

EntryProcessor крупным планом:



Рецепт достижения low-latency:

Направляйте вычисления к месту хранения данных.

Вместо того, чтобы подтягивать данные к месту проведения вычислений.

EntryProcessor изоляция транзакций:

- Изоляция транзакций в подавляющем большинстве случаев обеспечивается GRID-ом из коробки.
- Изоляцию можно случайно отключить, поэтому всегда лучше тестировать
  https://github.com/vladimir-bukhtoyarov/bucket4j/blob/4.10/doc-pages/production-jcache-checklist.md#verification-of-compatibility-with-particular-jcache-provider-is-your-responsibility

Interface EntryProcessor<K,V,T>:

Type Parameters:

K - the type of keys maintained by this cache

V - the type of cached values

T - the type of the return value

`public interface` **`EntryProcessor<K,V,T>`**

An invocable function that allows applications to perform compound operations on a `Cache.Entry` atomically, according the defined consistency of a `Cache`.

Any `Cache.Entry` mutations will not take effect until after the `process(MutableEntry, Object...)` method has completed execution.

If an exception is thrown by an `EntryProcessor`, a Caching Implementation must wrap any `Exception` thrown wrapped in an `EntryProcessorException`. If this occurs no mutations will be made to the `Cache.Entry`.

Implementations may execute `EntryProcessors` in situ, thus avoiding locking, round-trips and expensive network transfers.

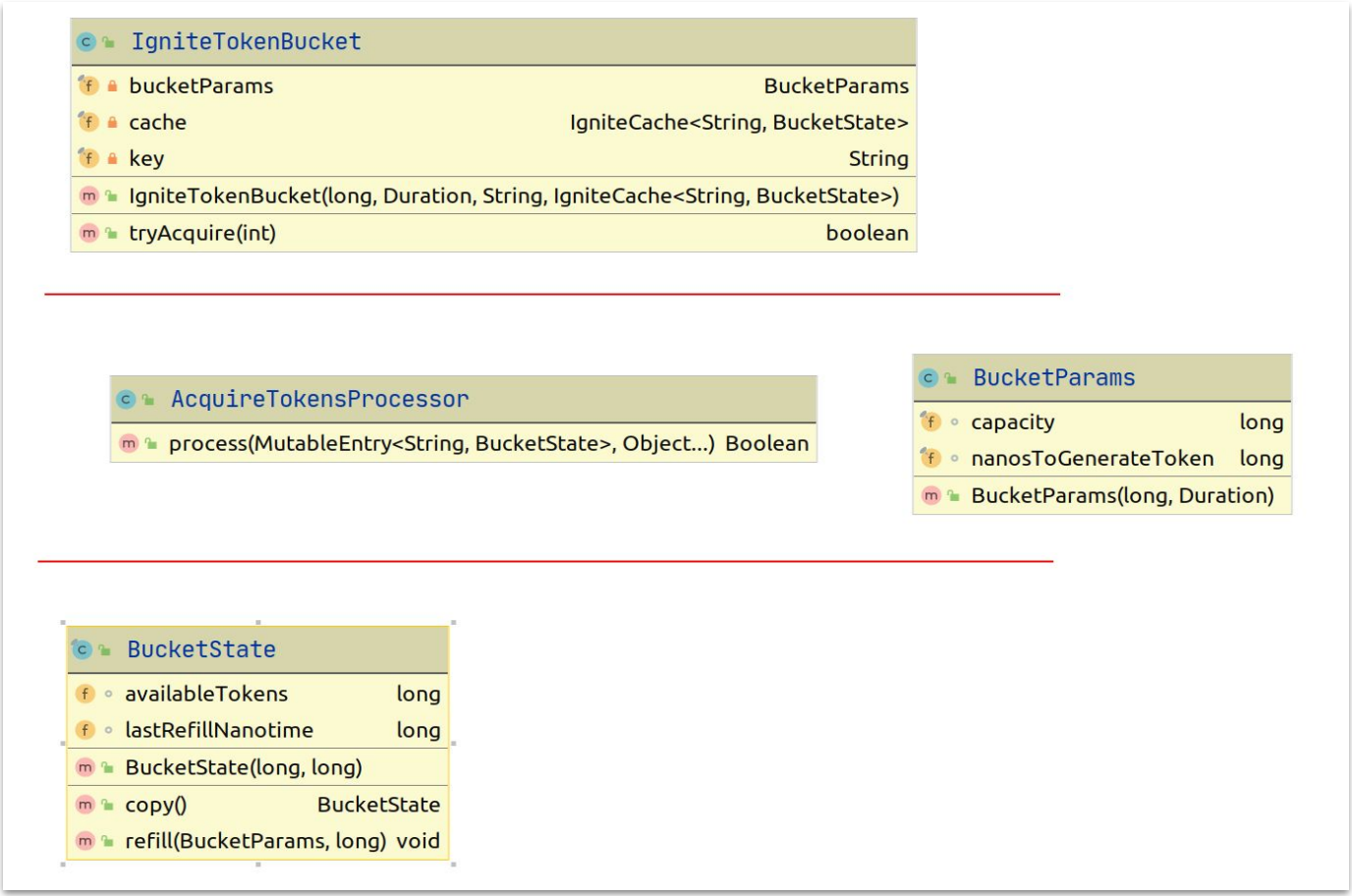
Do not use Redisson, and similar JCache adapters

Какую реализацию JSR-107 выбрать

- Hazelcast?
- Ignite?
- Infinispan?
- Coherence?

Да практически любую, **invoke(processor)** - это базовый функционал, примерно одинаково работающий в топовых реализациях.

IgniteTokenBucket диаграмма классов



DINS

IgniteTokenBucket состояние хранимое в кэше:

```
public final class BucketState implements Serializable {

    long availableTokens;
    long lastRefillNanotime;

    public BucketState(long availableTokens, long lastRefillNanotime) {
        this.availableTokens = availableTokens;
        this.lastRefillNanotime = lastRefillNanotime;
    }

    public BucketState copy() {
        return new BucketState(availableTokens, lastRefillNanotime);
    }

    public void refill(BucketParams params, long now) {
        long nanosSinceLastRefill = now - lastRefillNanotime;
        if (nanosSinceLastRefill <= params.nanosToGenerateToken) {
            return;
        }
        long tokensSinceLastRefill = nanosSinceLastRefill / params.nanosToGenerateToken;
        availableTokens = Math.min(params.capacity, availableTokens + tokensSinceLastRefill);
        lastRefillNanotime += tokensSinceLastRefill * params.nanosToGenerateToken;
    }

}
```

IgniteTokenBucket параметр для EntryProcessor:

```
public class BucketParams implements Serializable {

    final long capacity;
    final long nanosToGenerateToken;

    public BucketParams(long capacity, Duration period) {
        this.capacity = capacity;
        this.nanosToGenerateToken = period.toNanos() / capacity;
    }

}
```

IgniteTokenBucket реализация EntryProcessor:

```
public class AcquireTokensProcessor implements Serializable,
        EntryProcessor<String, BucketState, Boolean> {

    @Override
    public Boolean process(MutableEntry<String, BucketState> entry,
           Object... arguments) throws EntryProcessorException {
        final int tokensToConsume = (int) arguments[0];
        final BucketParams params = (BucketParams) arguments[1];
        long now = System.currentTimeMillis() * 1_000_000L;

        BucketState state;
        if (!entry.exists()) {
            state = new BucketState(params.capacity, now);
        } else {
            BucketState persistedState = entry.getValue();
            state = persistedState.copy();
            state.refill(params, now);
        }
        if (state.availableTokens < tokensToConsume) {
            return false;
        }
        state.availableTokens -= tokensToConsume;
        entry.setValue(state);
        return true;
    }

}
```

IgniteTokenBucket local part:

```
public class IgniteTokenBucket implements RateLimiter {

    private final BucketParams bucketParams;
    private final IgniteCache<String, BucketState> cache;
    private final String key;

    public IgniteTokenBucket(long permits, Duration period,
              String key, IgniteCache<String, BucketState> cache) {
        this.bucketParams = new BucketParams(permits, period);
        this.key = key;
        this.cache = cache;
    }

    @Override
    public boolean tryAcquire(int permits) {
        AcquireTokensProcessor processor = new AcquireTokensProcessor();
        return cache.invoke(key, processor, permits, bucketParams);
    }

}
```

IgniteTokenBucket решаем проблемы производительности:

- Проблема блокирующего IO.
- Проблема большого рэйта запросов на одном ключе.

IgniteTokenBucket blocking IO problem:

```
public class IgniteTokenBucket implements RateLimiter {

    private final BucketParams bucketParams;
    private final IgniteCache<String, BucketState> cache;
    private final String key;

    public IgniteTokenBucket(long permits, Duration period,
             String key, IgniteCache<String, BucketState> cache) {
        this.bucketParams = new BucketParams(permits, period);
        this.key = key;
        this.cache = cache;
    }

    @Override
    public boolean tryAcquire(int permits) {
        AcquireTokensProcessor processor = new AcquireTokensProcessor();
        return cache.invoke(key, processor, permits, bucketParams);
    }

}
```

IgniteAsyncTokenBucket:

```
public CompletableFuture<Boolean> tryAcquire(int numberTokens) {
    AcquireTokensProcessor entryProcessor = new AcquireTokensProcessor();
    IgniteFuture<Boolean> igniteFuture = cache.invokeAsync(key, entryProcessor, numberTokens, bucketParams);
    return convertFuture(igniteFuture);
}

private static <T> CompletableFuture<T> convertFuture(IgniteFuture<T> igniteFuture) {
    CompletableFuture<T> completableFuture = new CompletableFuture<>();
    igniteFuture.listen((IgniteInClosure<IgniteFuture<T>>) completedIgniteFuture -> {
        try {
            completableFuture.complete(completedIgniteFuture.get());
        } catch (Throwable t) {
            completableFuture.completeExceptionally(t);
        }
    });
    return completableFuture;
}
```

Проблема большого рэйта запросов на одно ключе:

JSR 107 реализации очень быстры. Но это заявление справедливо пока запросы хорошо распределены между ключами.

В случае когда все запросы приходится на один ключ, они сразу же превращаются в однопоточные, и не обеспечить ни приемлемого latency, ни throughput выше пары тысяч операций в секунду.

Используем батчинг на стороне клиента:

- Первый запрос всегда пропускаем на сервер.
- Пока запрос выполняется все последующие накапливаются в пачку.
- После завершения первого запроса, объединяем накопленные запросы и отправляем их на сервер как один запрос.
- Выполняем в цикле запросы на сервере. при этом прочитать состояние достаточно прочитать только для первого запроса, а сохранить только для последнего.

IgniteAsyncTokenBucket:

```
public class IgniteAsyncBatchingTokenBucket {

    private final BucketParams bucketParams;
    private final IgniteCache<String, BucketState> cache;
    private final String key;

    private final BatchHelper<Long, Boolean, List<Long>, List<Boolean>> batchHelper = BatchHelper.async(this::invokeBatch);

    private CompletableFuture<List<Boolean>> invokeBatch(List<Long> commands) {
        IgniteFuture<List<Boolean>> future = cache.invokeAsync(key, new BatchAcquireProcessor(), commands, bucketParams);
        return convertFuture(future);
    }

    public IgniteAsyncBatchingTokenBucket(long permits, Duration period, String key, IgniteCache<String, BucketState> cache) {
        this.bucketParams = new BucketParams(permits, period);
        this.key = key;
        this.cache = cache;
    }

    public CompletableFuture<Boolean> tryAcquire(long numberTokens) {
        return batchHelper.executeAsync(numberTokens);
    }
```

IgniteAsyncBatchingTokenBucket:

```
public class IgniteAsyncBatchingTokenBucket {

    private final BucketParams bucketParams;
    private final IgniteCache<String, BucketState> cache;
    private final String key;

    private final BatchHelper<Long, Boolean, List<Long>, List<Boolean>> batchHelper = BatchHelper.async(this::invokeBatch);

    private CompletableFuture<List<Boolean>> invokeBatch(List<Long> commands) {
        IgniteFuture<List<Boolean>> future = cache.invokeAsync(key, new BatchAcquireProcessor(), commands, bucketParams);
        return convertFuture(future);
    }

    public IgniteAsyncBatchingTokenBucket(long permits, Duration period, String key, IgniteCache<String, BucketState> cache) {
        this.bucketParams = new BucketParams(permits, period);
        this.key = key;
        this.cache = cache;
    }

    public CompletableFuture<Boolean> tryAcquire(long numberTokens) {
        return batchHelper.executeAsync(numberTokens);
    }
```

```java
public class BatchAcquireProcessor implements Serializable, EntryProcessor<String, BucketState, List<Boolean>> {

    @Override
    public List<Boolean> process(MutableEntry<String, BucketState> entry, Object... arguments) throws EntryProcessorException {
        final List<Long> tryConsumeCommands = (List<Long>) arguments[0];
        final BucketParams params = (BucketParams) arguments[1];
        long now = System.currentTimeMillis() * 1_000_000L;

        BucketState state;
        if (!entry.exists()) {
            state = new BucketState(params.capacity,  lastRefillNanotime: System.currentTimeMillis() * 1_000_000L);
        } else {
            BucketState persistedState = entry.getValue();
            state = persistedState.copy();
            state.refill(params, now);
        }

        // Execute batch
        List<Boolean> results = new ArrayList<>(tryConsumeCommands.size());
        long consumedTokens = 0;
        for (Long tokensToConsume : tryConsumeCommands) {
            if (state.availableTokens < tokensToConsume) {
                results.add(false);
            } else {
                state.availableTokens -= tokensToConsume;
                results.add(true);
                consumedTokens += tokensToConsume;
            }
        }

        // save results if something was consumed
        if (consumedTokens > 0) {
            entry.setValue(state);
        }

        return results;
    }

}
```

Итоги:

- Научились имплементировать Distributed Token Bucket с помощью Apache Ignite.
- Обошли несколько перфоманс проблем.
- Поняли что Ignite это больше чем просто кэш второго уровня для Hibernate

Полезные ссылки:

- https://en.wikipedia.org/wiki/Token_bucket
- https://github.com/bbeck/token-bucket
- https://github.com/vladimir-bukhtoyarov/bucket4j
- https://linkmeup.gitbook.io/sdsm/15.-qos/7.-ogranichenie-skorosti/4-mekhanizmy-leaky-bucket-i-token-bucket/1-algoritm-token-bucket
- https://engineering.grab.com/quotas-service
- https://hazelcast.com/blog/an-easy-performance-improvement-with-entryprocessor/
- https://www.youtube.com/watch?v=6Xc8HD1pSNU

Thank you

Vladimir Bukhtoyarov

jsecoder@mail.ru

http://github.com/vladimir-bukhtoyarov/token-bucket-demo