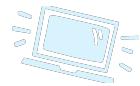




# Реализуем распределённый Rate-Limiter на Apache Ignite



1 / 14



...



# О докладчике:

---

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

## План доклада

- Общий обзор задач ограничения пропускной способности.
- Специфика rate limiting в распределенной среде.
- Обзор алгоритма Token Bucket.
- Реализуем распределенный Token Bucket с использованием технологии Apache Ignite.
- Устраним ошибки производительности получившегося 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 БОТ 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);  
}
```

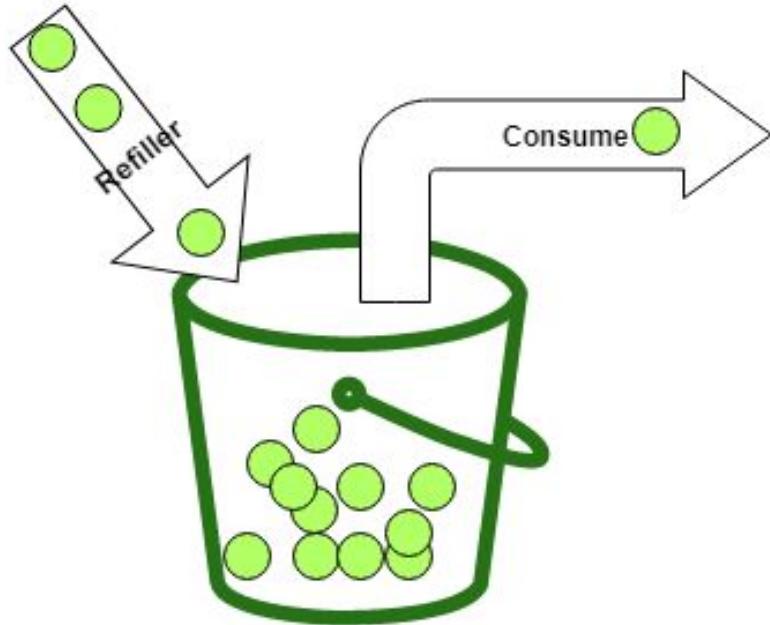


# Выбор алгоритма RateLimiter

- 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://en.wikipedia.org/wiki/Token\\_bucket](https://en.wikipedia.org/wiki/Token_bucket)

# TokenBucket реализация

```
public class LockFreeTokenBucket implements RateLimiter {

    private final BucketParams params;
    private final AtomicReference<BucketState> stateReference;

    public LockFreeTokenBucket(long permits, Duration period) {
        this.params = new BucketParams(permits, period);
        this.stateReference = new AtomicReference<>(new BucketState(params, nanoTime()));
    }

    public boolean tryAcquire(int permits) {
        while (true) {
            long nanoTime = nanoTime();
            BucketState previousState = stateReference.get();
            BucketState newState = new BucketState(previousState);
            newState.refill(params, nanoTime);
            if (newState.availableTokens < permits) {
                return false;
            }
            newState.availableTokens -= permits;
            if (stateReference.compareAndSet(previousState, newState)) {
                return true;
            }
        }
    }
}
```

# TokenBucket реализация

```
public class LockFreeTokenBucket implements RateLimiter {

    private final BucketParams params;
    private final AtomicReference<BucketState> stateReference;

    public LockFreeTokenBucket(long permits, Duration period) {
        this.params = new BucketParams(permits, period);
        this.stateReference = new AtomicReference<>(new BucketState(params, nanoTime()));
    }

    public boolean tryAcquire(int permits) {
        while (true) {
            long nanoTime = nanoTime();
            BucketState previousState = stateReference.get();
            BucketState newState = new BucketState(previousState);
            newState.refill(params, nanoTime);
            if (newState.availableTokens < permits) {
                return false;
            }
            newState.availableTokens -= permits;
            if (stateReference.compareAndSet(previousState, newState)) {
                return true;
            }
        }
    }
}
```

# TokenBucket параметры

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

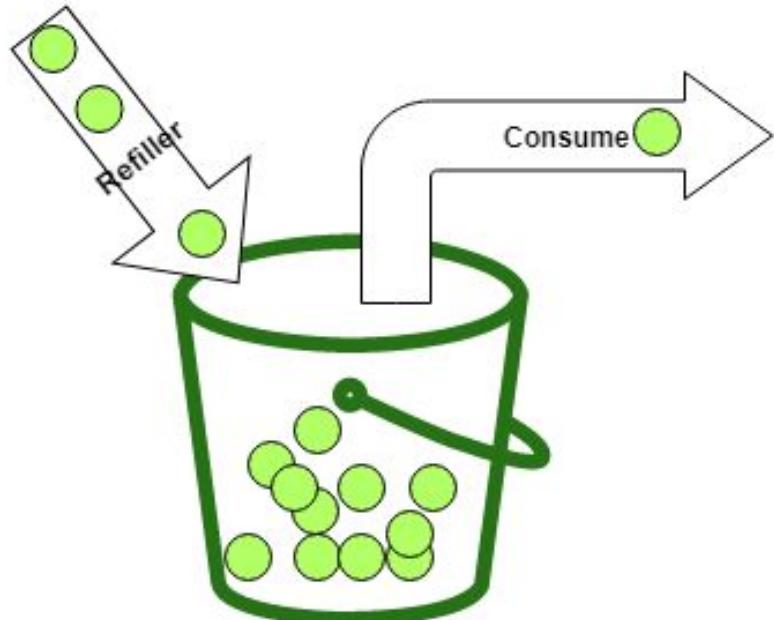
# TokenBucket состояния

```
public final class BucketState {  
  
    long availableTokens;  
    long lastRefillNanoTime;  
  
    public BucketState(BucketParams params, long nanoTime) {  
        this.lastRefillNanoTime = nanoTime;  
        this.availableTokens = params.capacity;  
    }  
  
    public BucketState(BucketState other) {  
        this.lastRefillNanoTime = other.lastRefillNanoTime;  
        this.availableTokens = other.availableTokens;  
    }  
  
    public void refill(BucketParams params, long nanoTime) {  
        long nanosSinceLastRefill = nanoTime - this.lastRefillNanoTime;  
        if (nanosSinceLastRefill <= params.nanosToGenerateToken) {  
            return;  
        }  
        long tokensSinceLastRefill = nanosSinceLastRefill / params.nanosToGenerateToken;  
        availableTokens = Math.min(params.capacity, availableTokens + tokensSinceLastRefill);  
        lastRefillNanoTime += tokensSinceLastRefill * params.nanosToGenerateToken;  
    }  
}
```

# TokenBucket состояния

```
public final class BucketState {  
  
    long availableTokens;  
    long lastRefillNanoTime;  
  
    public BucketState(BucketParams params, long nanoTime) {  
        this.lastRefillNanoTime = nanoTime;  
        this.availableTokens = params.capacity;  
    }  
  
    public BucketState(BucketState other) {  
        this.lastRefillNanoTime = other.lastRefillNanoTime;  
        this.availableTokens = other.availableTokens;  
    }  
  
    public void refill(BucketParams params, long nanoTime) {  
        long nanosSinceLastRefill = nanoTime - this.lastRefillNanoTime;  
        if (nanosSinceLastRefill <= params.nanosToGenerateToken) {  
            return;  
        }  
        long tokensSinceLastRefill = nanosSinceLastRefill / params.nanosToGenerateToken;  
        availableTokens = Math.min(params.capacity, availableTokens + tokensSinceLastRefill);  
        lastRefillNanoTime += tokensSinceLastRefill * params.nanosToGenerateToken;  
    }  
}
```

# TokenBucket состояние



[https://en.wikipedia.org/wiki/Token\\_bucket](https://en.wikipedia.org/wiki/Token_bucket)

# TokenBucket состояния

```
public final class BucketState {  
  
    long availableTokens;  
    long lastRefillNanoTime;  
  
    public BucketState(BucketParams params, long nanoTime) {  
        this.lastRefillNanoTime = nanoTime;  
        this.availableTokens = params.capacity;  
    }  
  
    public BucketState(BucketState other) {  
        this.lastRefillNanoTime = other.lastRefillNanoTime;  
        this.availableTokens = other.availableTokens;  
    }  
  
    public void refill(BucketParams params, long nanoTime) {  
        long nanosSinceLastRefill = nanoTime - this.lastRefillNanoTime;  
        if (nanosSinceLastRefill <= params.nanosToGenerateToken) {  
            return;  
        }  
        long tokensSinceLastRefill = nanosSinceLastRefill / params.nanosToGenerateToken;  
        availableTokens = Math.min(params.capacity, availableTokens + tokensSinceLastRefill)  
        lastRefillNanoTime += tokensSinceLastRefill * params.nanosToGenerateToken;  
    }  
}
```

# TokenBucket состояния

```
public final class BucketState {  
  
    long availableTokens;  
    long lastRefillNanoTime;  
  
    public BucketState(BucketParams params) {  
        this.lastRefillNanoTime = System.nanoTime();  
        this.availableTokens = params.capacity;  
    }  
  
    public BucketState(BucketState other) {  
        this.lastRefillNanoTime = other.lastRefillNanoTime;  
        this.availableTokens = other.availableTokens;  
    }  
  
    public void refill(BucketParams params, long nanoTime) {  
        long nanosSinceLastRefill = nanoTime - this.lastRefillNanoTime;  
        if (nanosSinceLastRefill <= params.nanosToGenerateToken) {  
            return;  
        }  
        long tokensSinceLastRefill = nanosSinceLastRefill / params.nanosToGenerateToken;  
        availableTokens = Math.min(params.capacity, availableTokens + tokensSinceLastRefill);  
        lastRefillNanoTime += tokensSinceLastRefill * params.nanosToGenerateToken;  
    }  
}
```

# TokenBucket состояния

```
public final class BucketState {  
  
    long availableTokens;  
    long lastRefillNanoTime;  
  
    public BucketState(BucketParams params, long nanoTime) {  
        this.lastRefillNanoTime = nanoTime;  
        this.availableTokens = params.capacity;  
    }  
  
    public BucketState(BucketState other) {  
        this.lastRefillNanoTime = other.lastRefillNanoTime;  
        this.availableTokens = other.availableTokens;  
    }  
  
    public void refill(BucketParams params, long nanoTime) {  
        long nanosSinceLastRefill = nanoTime - this.lastRefillNanoTime;  
        if (nanosSinceLastRefill <= params.nanosToGenerateToken) {  
            return;  
        }  
        long tokensSinceLastRefill = nanosSinceLastRefill / params.nanosToGenerateToken;  
        availableTokens = Math.min(params.capacity, availableTokens + tokensSinceLastRefill);  
        lastRefillNanoTime += tokensSinceLastRefill * params.nanosToGenerateToken;  
    }  
}
```

# TokenBucket состояния

```
public final class BucketState {  
  
    long availableTokens;  
    long lastRefillNanoTime;  
  
    public BucketState(BucketParams params, long nanoTime) {  
        this.lastRefillNanoTime = nanoTime;  
        this.availableTokens = params.capacity;  
    }  
  
    public BucketState(BucketState other) {  
        this.lastRefillNanoTime = other.lastRefillNanoTime;  
        this.availableTokens = other.availableTokens;  
    }  
  
    public void refill(BucketParams params, long nanoTime) {  
        long nanosSinceLastRefill = nanoTime - this.lastRefillNanoTime;  
        if (nanosSinceLastRefill <= params.nanosToGenerateToken) {  
            return;  
        }  
        long tokensSinceLastRefill = nanosSinceLastRefill / params.nanosToGenerateToken;  
        availableTokens = Math.min(params.capacity, availableTokens + tokensSinceLastRefill);  
        lastRefillNanoTime += tokensSinceLastRefill * params.nanosToGenerateToken;  
    }  
}
```

# TokenBucket реализация

```
public class LockFreeTokenBucket implements RateLimiter {

    private final BucketParams params;
    private final AtomicReference<BucketState> stateReference;

    public LockFreeTokenBucket(long permits, Duration period) {
        this.params = new BucketParams(permits, period);
        this.stateReference = new AtomicReference<>(new BucketState(params));
    }

    public boolean tryAcquire(int permits) {
        while (true) {
            long nanoTime = System.nanoTime();
            BucketState previousState = stateReference.get();
            BucketState newState = new BucketState(previousState);
            newState.refill(params, nanoTime);
            if (newState.availableTokens < permits) {
                return false;
            }
            newState.availableTokens -= permits;
            if (stateReference.compareAndSet(previousState, newState)) {
                return true;
            }
        }
    }
}
```

# TokenBucket реализация

```
public class LockFreeTokenBucket implements RateLimiter {

    private final BucketParams params;
    private final AtomicReference<BucketState> stateReference;

    public LockFreeTokenBucket(long permits, Duration period) {
        this.params = new BucketParams(permits, period);
        this.stateReference = new AtomicReference<>(new BucketState(params, nanoTime()));
    }

    public boolean tryAcquire(int permits) {
        while (true) {
            long nanoTime = nanoTime();
            BucketState previousState = stateReference.get();
            BucketState newState = new BucketState(previousState);
            newState.refill(params, nanoTime);
            if (newState.availableTokens < permits) {
                return false;
            }
            newState.availableTokens -= permits;
            if (stateReference.compareAndSet(previousState, newState)) {
                return true;
            }
        }
    }
}
```

# TokenBucket реализация

```
public class LockFreeTokenBucket implements RateLimiter {

    private final BucketParams params;
    private final AtomicReference<BucketState> stateReference;

    public LockFreeTokenBucket(long permits, Duration period) {
        this.params = new BucketParams(permits, period);
        this.stateReference = new AtomicReference<>(new BucketState(params, nanoTime()));
    }

    public boolean tryAcquire(int permits) {
        while (true) {
            long nanoTime = nanoTime();
            BucketState previousState = stateReference.get();
            BucketState newState = new BucketState(previousState);
            newState.refill(params, nanoTime);
            if (newState.availableTokens < permits) {
                return false;
            }
            newState.availableTokens -= permits;
            if (stateReference.compareAndSet(previousState, newState)) {
                return true;
            }
        }
    }
}
```

# TokenBucket реализация

```
public class LockFreeTokenBucket implements RateLimiter {

    private final BucketParams params;
    private final AtomicReference<BucketState> stateReference;

    public LockFreeTokenBucket(long permits, Duration period) {
        this.params = new BucketParams(permits, period);
        this.stateReference = new AtomicReference<>(new BucketState(params, nanoTime()));
    }

    public boolean tryAcquire(int permits) {
        while (true) {
            long nanoTime = nanoTime();
            BucketState previousState = stateReference.get();
            BucketState newState = new BucketState(previousState);
            newState.refill(params, nanoTime);
            if (newState.availableTokens < permits) {
                return false;
            }
            newState.availableTokens -= permits;
            if (stateReference.compareAndSet(previousState, newState)) {
                return true;
            }
        }
    }
}
```

# TokenBucket реализация

```
public class LockFreeTokenBucket implements RateLimiter {

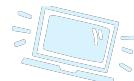
    private final BucketParams params;
    private final AtomicReference<BucketState> stateReference;

    public LockFreeTokenBucket(long permits, Duration period) {
        this.params = new BucketParams(permits, period);
        this.stateReference = new AtomicReference<>(new BucketState(params, nanoTime()));
    }

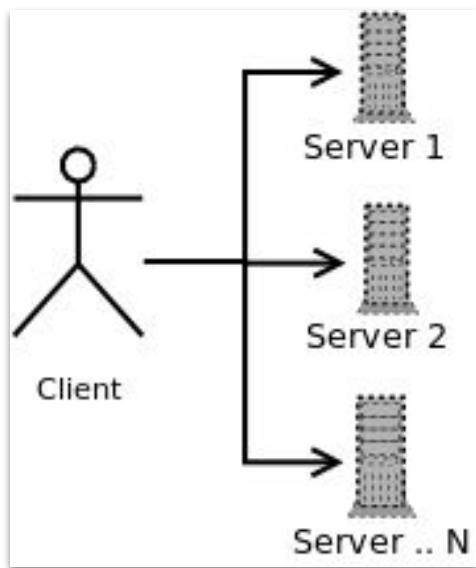
    public boolean tryAcquire(int permits) {
        while (true) {
            long nanoTime = nanoTime();
            BucketState previousState = stateReference.get();
            BucketState newState = new BucketState(previousState);
            newState.refill(params, nanoTime);
            if (newState.availableTokens < permits) {
                return false;
            }
            newState.availableTokens -= permits;
            if (stateReference.compareAndSet(previousState, newState)) {
                return true;
            }
        }
    }
}
```



# Переходим к распределенной реализации



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



При 10 серверах в кластере, нельзя просто взять глобальный лимит 200 req/min и преобразовать его в 10 локальных лимитов 20 req/min:

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

# Классификация подходов репликации состояния для распределенного RateLimiter

- Централизованный синхронный:

<https://github.com/mokies/ratelimitj>

<https://github.com/vladimir-bukhtoyarov/bucket4j>

<https://resilience4j.readme.io/docs/ratelimiter>

- Централизованный асинхронный:

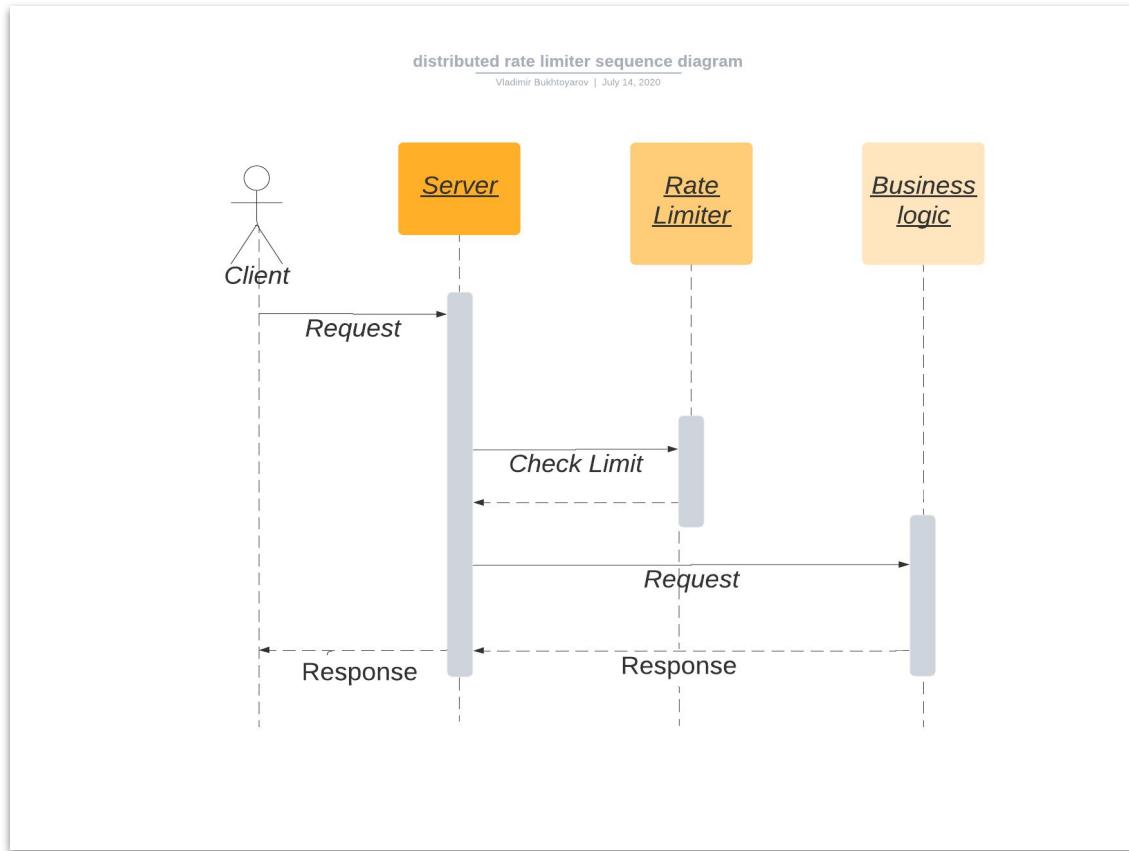
<https://engineering.grab.com/quotas-service>

<https://www.mailgun.com/blog/governator-cloud-native-distributed-rate-limiting-microservices/>

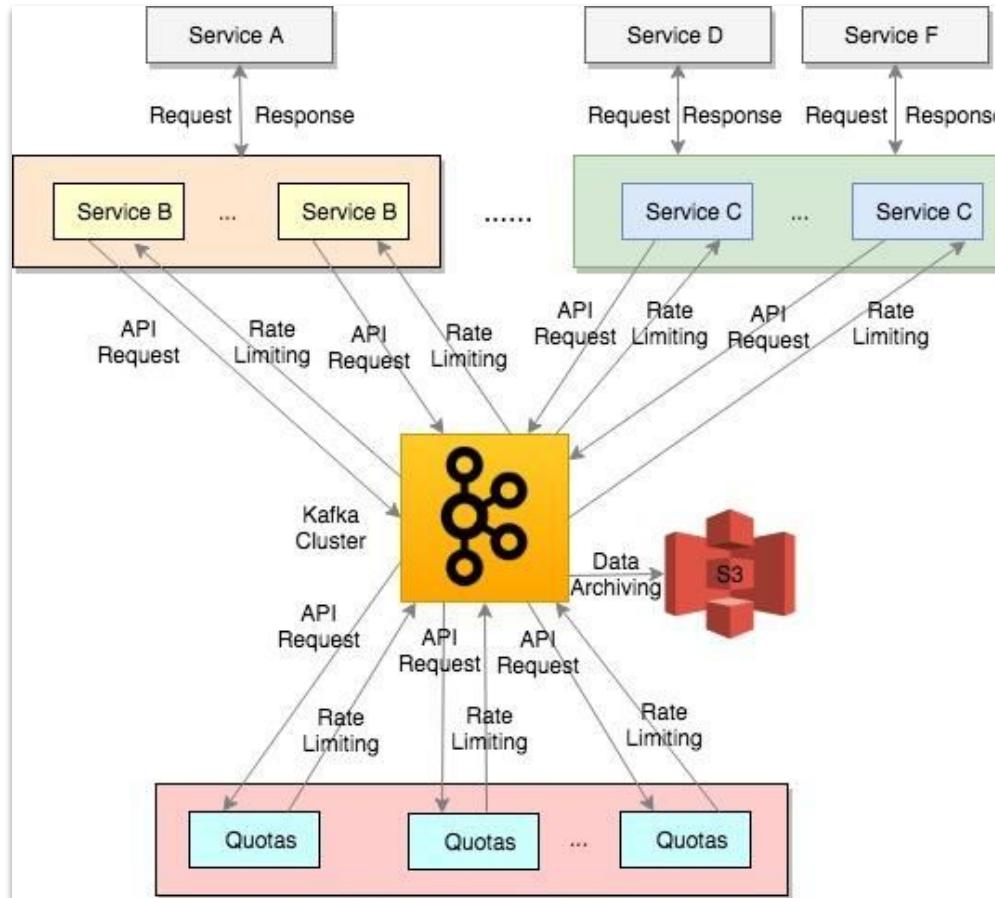
- Децентрализованный:

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

# Централизованный синхронный RateLimiter



# Централизованный асинхронный RateLimiter



# Классификация подходов репликации состояния для распределенного RateLimiter

Способ	Единая точка хранения состояния	Точность	Ресурсоемкость
Централизованный синхронный	Да	Абсолютная	Высокая
Централизованный асинхронный	Да	Возможны задержки	Средняя
Децентрализованный	Нет	???	Низкая



# Distributed TokenBucket выбор технологии для реализации:

- Обзор требований.
- Муки выбора технологий.



## 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.

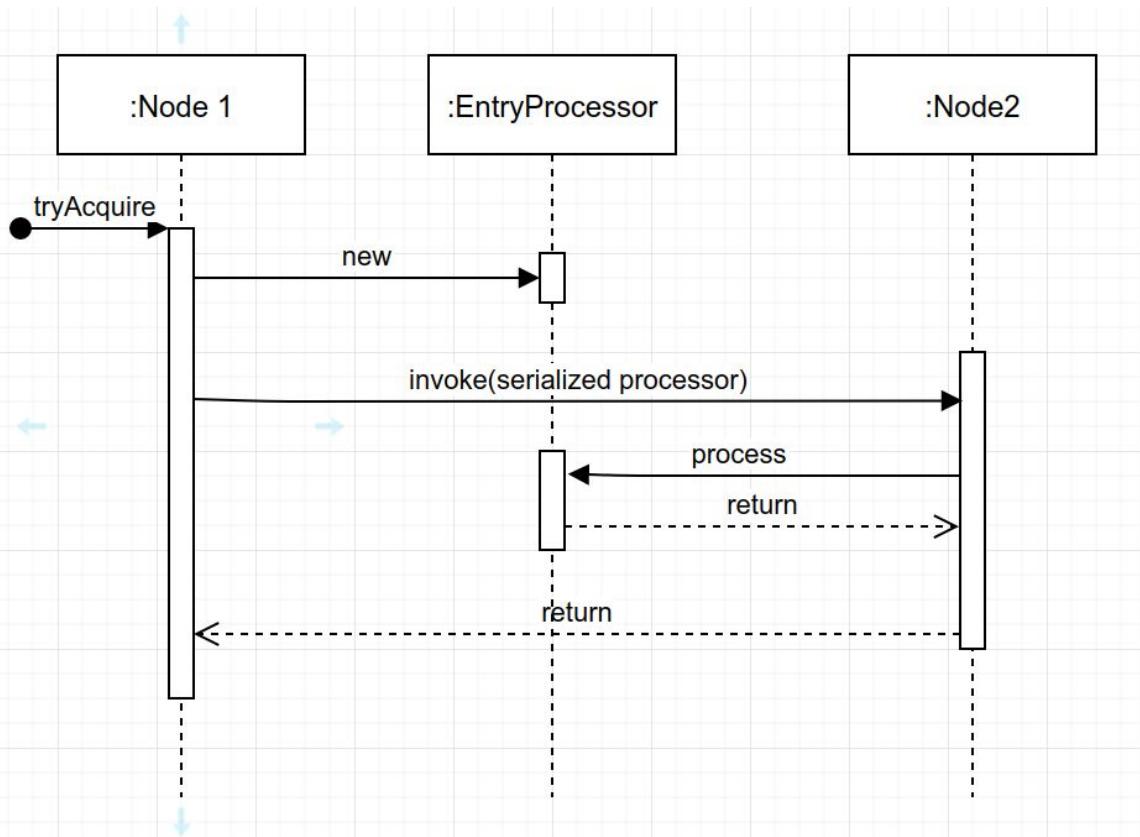
## Чего нам не требуется:

- Хранение данных на дисках, достаточно простой репликации.
- 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. Нужно писать на языках отличных от Java.
- 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.

# Interface EntryProcessor<K,V,T>

```
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.

# 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>

# Какую реализацию JSR-107 выбрать

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

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

# IgniteTokenBucket

```
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

```
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 реализация EntryProcessor

```
public class AcquireTokensProcessor implements Serializable, EntryProcessor<String, BucketState, Boolean> {  
    @Override  
    public Boolean process(MutableEntry<String, BucketState> entry, Object... args)  
        throws EntryProcessorException {  
        final int tokensToConsume = (int) args[0];  
        final BucketParams params = (BucketParams) args[1];  
        long now = TimeUnit.MILLISECONDS.toNanos(System.currentTimeMillis());  
  
        BucketState state = entry.exists() ? new BucketState(entry.getValue()) : new BucketState(params, now);  
        state.refill(params, now);  
  
        if (state.availableTokens < tokensToConsume) {  
            return false;  
        }  
        state.availableTokens -= tokensToConsume;  
        entry.setValue(state);  
        return true;  
    }  
}
```

# IgniteTokenBucket реализация EntryProcessor

```
public class AcquireTokensProcessor implements Serializable, EntryProcessor<String, BucketState, Boolean> {
    @Override
    public Boolean process(MutableEntry<String, BucketState> entry, Object... args)
        throws EntryProcessorException {
        final int tokensToConsume = (int) args[0];
        final BucketParams params = (BucketParams) args[1];
        long now = TimeUnit.MILLISECONDS.toNanos(System.currentTimeMillis());
        BucketState state = entry.exists() ? new BucketState(entry.getValue()) : new BucketState(params, now);
        state.refill(params, now);

        if (state.availableTokens < tokensToConsume) {
            return false;
        }
        state.availableTokens -= tokensToConsume;
        entry.setValue(state);
        return true;
    }
}
```

# IgniteTokenBucket реализация EntryProcessor

```
public class AcquireTokensProcessor implements Serializable, EntryProcessor<String, BucketState, Boolean> {  
    @Override  
    public Boolean process(MutableEntry<String, BucketState> entry, Object... args)  
        throws EntryProcessorException {  
        final int tokensToConsume = (int) args[0];  
        final BucketParams params = (BucketParams) args[1];  
        long now = TimeUnit.MILLISECONDS.toNanos(System.currentTimeMillis());  
  
        BucketState state = entry.exists() ? new BucketState(entry.getValue()) : new BucketState(params, now);  
        state.refill(params, now);  
  
        if (state.availableTokens < tokensToConsume) {  
            return false;  
        }  
        state.availableTokens -= tokensToConsume;  
        entry.setValue(state);  
        return true;  
    }  
}
```

# IgniteTokenBucket реализация EntryProcessor

```
public class AcquireTokensProcessor implements Serializable, EntryProcessor<String, BucketState, Boolean> {
    @Override
    public Boolean process(MutableEntry<String, BucketState> entry, Object... args)
        throws EntryProcessorException {
        final int tokensToConsume = (int) args[0];
        final BucketParams params = (BucketParams) args[1];
        long now = TimeUnit.MILLISECONDS.toNanos(System.currentTimeMillis());

        BucketState state = entry.exists() ? new BucketState(entry.getValue()) : new BucketState(params, now);
        state.refill(params, now);

        if (state.availableTokens < tokensToConsume) {
            return false;
        }

        state.availableTokens -= tokensToConsume;
        entry.setValue(state);
        return true;
    }
}
```

# IgniteTokenBucket реализация EntryProcessor

```
public class AcquireTokensProcessor implements Serializable, EntryProcessor<String, BucketState, Boolean> {  
    @Override  
    public Boolean process(MutableEntry<String, BucketState> entry, Object... args)  
        throws EntryProcessorException {  
        final int tokensToConsume = (int) args[0];  
        final BucketParams params = (BucketParams) args[1];  
        long now = TimeUnit.MILLISECONDS.toNanos(System.currentTimeMillis());  
  
        BucketState state = entry.exists() ? new BucketState(entry.getValue()) : new BucketState(params, now);  
        state.refill(params, now);  
  
        if (state.availableTokens < tokensToConsume) {  
            return false;  
        }  
        state.availableTokens -= tokensToConsume;  
        entry.setValue(state);  
        return true;  
    }  
}
```



# igniteTokenBucket решаем проблемы производительности:

- Проблема блокирующего IO.
- Проблема большого рейтa 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 обеспечивают 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 состояние достаточно прочитать только для первого запроса, а сохранить только для последнего.

# 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);  
    }  
}
```

```
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 nanoTime = TimeUnit.MILLISECONDS.toNanos(System.currentTimeMillis());  
        BucketState state = entry.exists() ? new BucketState(entry.getValue()) : new BucketState(params, nanoTime);  
        state.refill(params, nanoTime);  
        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;  
            }  
        }  
        if (consumedTokens > 0) {  
            entry.setValue(state);  
        }  
        return results;  
    }  
}
```

```
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 nanoTime = TimeUnit.MILLISECONDS.toNanos(System.currentTimeMillis());  
        BucketState state = entry.exists() ? new BucketState(entry.getValue()) : new BucketState(params, nanoTime);  
        state.refill(params, nanoTime);  
        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;  
            }  
        }  
        if (consumedTokens > 0) {  
            entry.setValue(state);  
        }  
        return results;  
    }  
}
```

```
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 nanoTime = TimeUnit.MILLISECONDS.toNanos(System.currentTimeMillis());  
        BucketState state = entry.exists() ? new BucketState(entry.getValue()) : new BucketState(params, nanoTime);  
        state.refill(params, nanoTime);  
        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;  
            }  
        }  
        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://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>

