DINS®

# Реализуем алгоритм “Token Bucket” на Java

DINS®

# О докладчике:

- 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 комичу в OpenSource проекты.

## 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 2200 проектов по данным github.
- Используется в JHipster, Armeria, Jobby, Kubernatess Java Client.

# План доклада:

- Общий обзор задач ограничение пропускной способности.
- Примеры типичных проблем при реализации троттлинга.
- Обзор алгоритма Token Bucket.
- Реализуем Lock-Free Token Bucket на Java.
- Реализуем распределенную реализацию Token Bucket с использованием Apache Ignite.

# Типовые случаи, когда мы сталкиваемся с необходимостью ограничивать пропускную способность:

- Защита нашей системы от перегрузки порождаемой запросами приходящими извне.
- Защита сторонних систем от нагрузки, которую порождает наша система.
- Реализация контрактных обязательств.
- Fraud Detection.

Большие сложные системы обычно сталкиваются с комбинацией нескольких юзкейсов одновременно, и  реализуют защиту как на аппаратном уровне, так и на уровне кода приложения. Выбор между аппаратным и программным троттлингом обычно очевиден и определяется характером решаемой задачи.

# Защищаем API социальной сети

```
@RequestMapping(method = RequestMethod.POST, value = "/advertising/message")
public void sendAdvertisingMessages(MessageBatch messages) throws RateLimitExceedException {
    int requiredPermits = messages.estimmateCost();
    RateLimiter rateLimiter = SecurityService.getRateLimiter();
    if (rateLimiter.tryAcquire(requiredPermits)) {
        throw new RateLimitExceedException();
    }
    messageService.send(messages);
}
```

# Минимально необходимый интерфейс:

```
public interface RateLimiter {

    boolean tryAcquire(int permits);

}
```

# Реализуем самодельный RateLimiter:

- Пример кода.
- Разбор проблем.

# Простой RateLimiter структуры данных

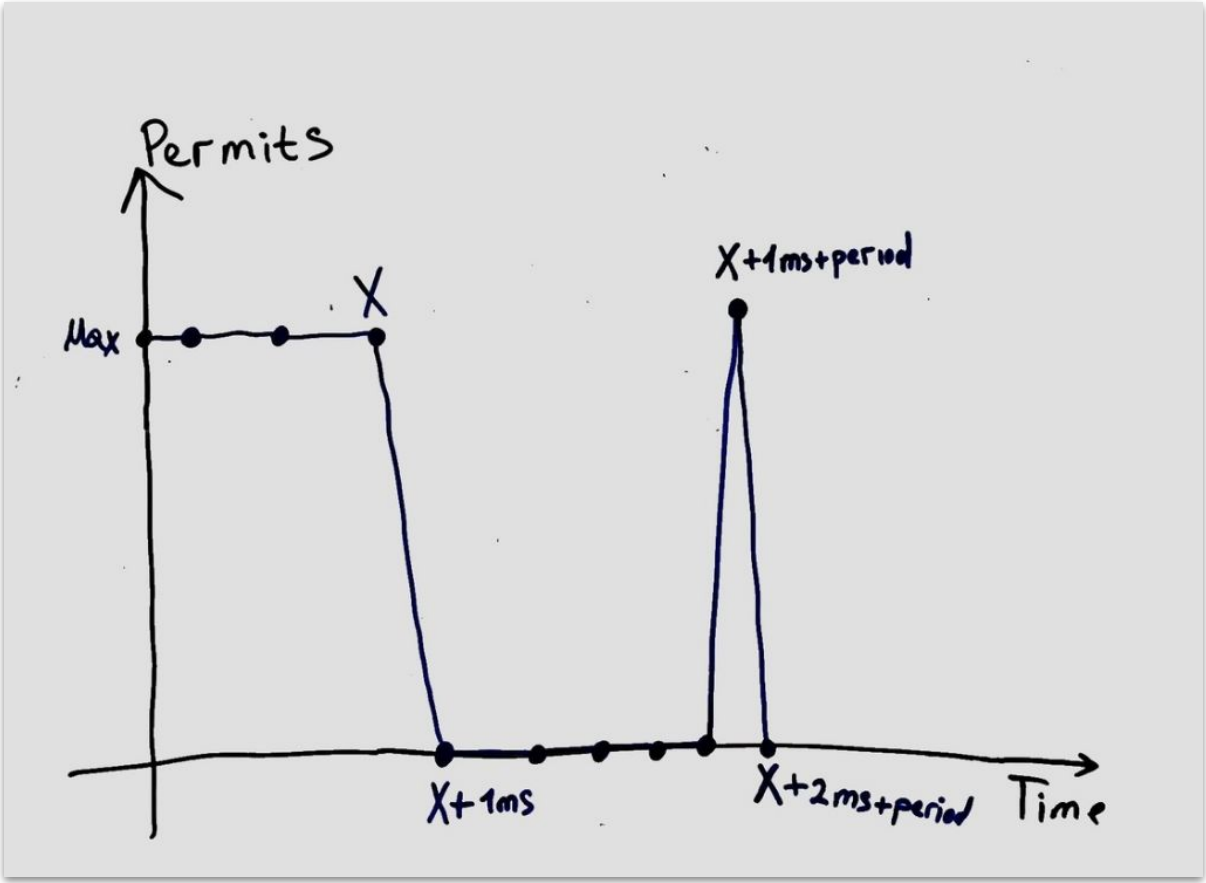
```java
public class SimpleRateLimiter implements RateLimiter {

    public SimpleRateLimiter(long permits, Duration period) {
        this.availablePermits = permits;
        this.periodNanos = period.toNanos();
    }

    private final long periodNanos;
    private long availablePermits;
    private final LinkedList<IssuedPermits> issuedTokens = new LinkedList<>();

    private static final class IssuedPermits {
        private final long permits;
        private final long expirationNanotime;

        private IssuedPermits(long permits, long expirationNanotime) {
            this.permits = permits;
            this.expirationNanotime = expirationNanotime;
        }
    }
```

# Простой RateLimiter алгоритм:

```java
@Override
synchronized public boolean tryAcquire(int numberTokens) {
    long nanoTime = System.nanoTime();
    clearPreviouslyIssuedPermits(nanoTime);

    if (availablePermits < numberTokens) {
        // has no requested permits
        return false;
    } else {
        long expirationNanoTime = nanoTime + periodNanos;
        issuedTokens.addLast(new IssuedPermits(numberTokens, expirationNanoTime));
        availablePermits -= numberTokens;
        return true;
    }
}

private void clearPreviouslyIssuedPermits(long currentNanotime) {
    while (!issuedTokens.isEmpty()) {
        IssuedPermits issue = issuedTokens.getFirst();
        if (currentNanotime > issue.expirationNanotime) {
            availablePermits += issue.permits;
            issuedTokens.removeFirst();
        } else {
            return;
        }
    }
}
```

```
long nanoTime = System.nanoTime()
```

# SimpleRateLimiter проблема Local Burst:



# SimpleRateLimiter проблема Local Burst:

При целевых параметрах 1000 permits / 1 second
Рэйт может достигнуть 2000 permits / 1,000000001 second

# SimpleRateLimiter решение проблемы Local Burst:

Целевые параметры
**M** permits / **N** period

Могут быть достигнуты через переформулирование
**M/2** permits / **N/2** period

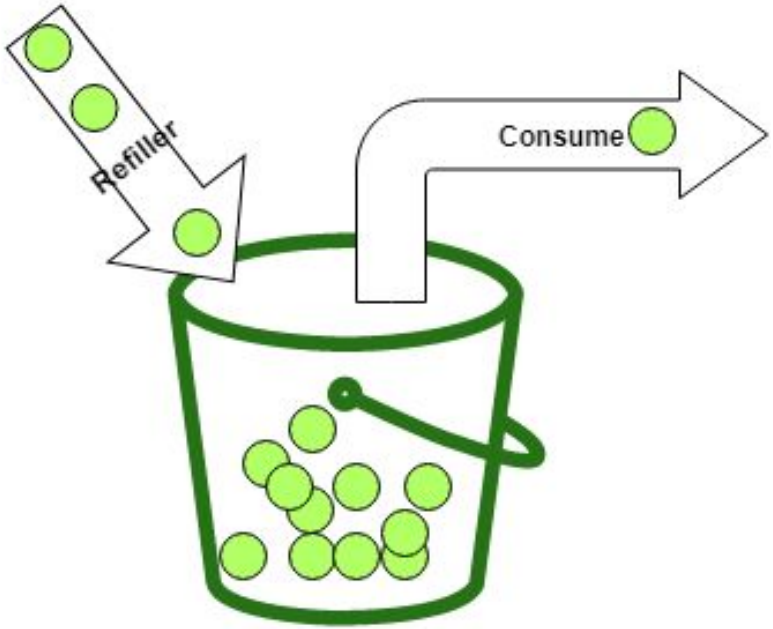
**Пример:**
1000 permits / 1 час заменить на 500 permits /30 минут

# Простой RateLimiter проблема потребления памяти:

```
public class SimpleRateLimiter implements RateLimiter {

    public SimpleRateLimiter(long permits, Duration period) {
        this.availablePermits = permits;
        this.periodNanos = period.toNanos();
    }

    private final long periodNanos;
    private long availablePermits;
    private final LinkedList<IssuedPermits> issuedTokens = new LinkedList<>();
```

# Ищем алгоритм с фиксированной памятью:

# Token Bucket: алгоритм с фиксированной памятью




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

Потребляемая память:

- Capacity ведра - Long 8 байт.
- Текущее число токенов в ведре - Long 8 байт.
- Число наносекунд на генерацию одного токена - Long 8 байт.

Итого 24 байта, в Managed языках ещё 16 байт на заголовок объекта.



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

# Реализуем Token Bucket на Java:

- Первоначальная реализация.
- Разбор проблем.

# TokenBucket структуры данных

```
public class StraightforwardTokenBucket implements RateLimiter {

    public StraightforwardTokenBucket(long permits, Duration period,
              ScheduledExecutorService scheduler) {
        this.capacity = permits;
        this.availableTokens = capacity;
        long nanosToGenerateToken = period.toNanos() / permits;
        scheduler.scheduleAtFixedRate(this::addToken,
                nanosToGenerateToken, nanosToGenerateToken, TimeUnit.NANOSECONDS
        );
    }

    private final long capacity;
    private long availableTokens;
```

# TokenBucket алгоритм:

```
private final long capacity;
private long availableTokens;

synchronized private void addToken() {
    availableTokens = Math.min(capacity, availableTokens + 1);
}

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

# TokenBucket проблема потребления CPU:

```
public StraightforwardTokenBucket(long permits, Duration period,
          ScheduledExecutorService scheduler) {
    this.capacity = permits;
    this.availableTokens = capacity;
    long nanosToGenerateToken = period.toNanos() / permits;
    scheduler.scheduleAtFixedRate(this::addToken,
            nanosToGenerateToken, nanosToGenerateToken, TimeUnit.NANOSECONDS
    );
}
```

# SchedulerFreeTokenBucket структуры 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;
    }
```

# SchedulerFreeTokenBucket 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;
}
```

# SchedulerFreeTokenBucket проблема блокировок

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

# LockFreeTokenBucket структура данных:

```
public class LockFreeTokenBucket {

    private final long capacity;
    private final long nanosToGenerateToken;
    private final AtomicReference<State> stateReference;

    private static final class State {

        private long availableTokens;
        private long lastRefillNanotime;

    }

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

        State initialState = new State();
        initialState.lastRefillNanotime = System.nanoTime();
        initialState.availableTokens = permits;

        this.stateReference = new AtomicReference<>(initialState);
    }
```

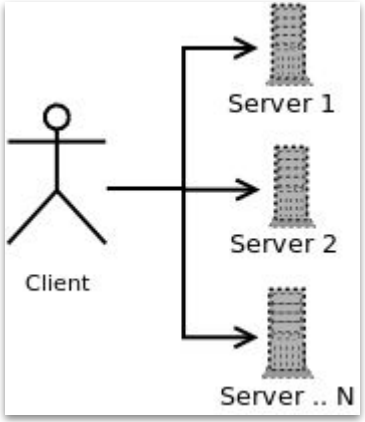
# LockFreeTokenBucket алгоритм:

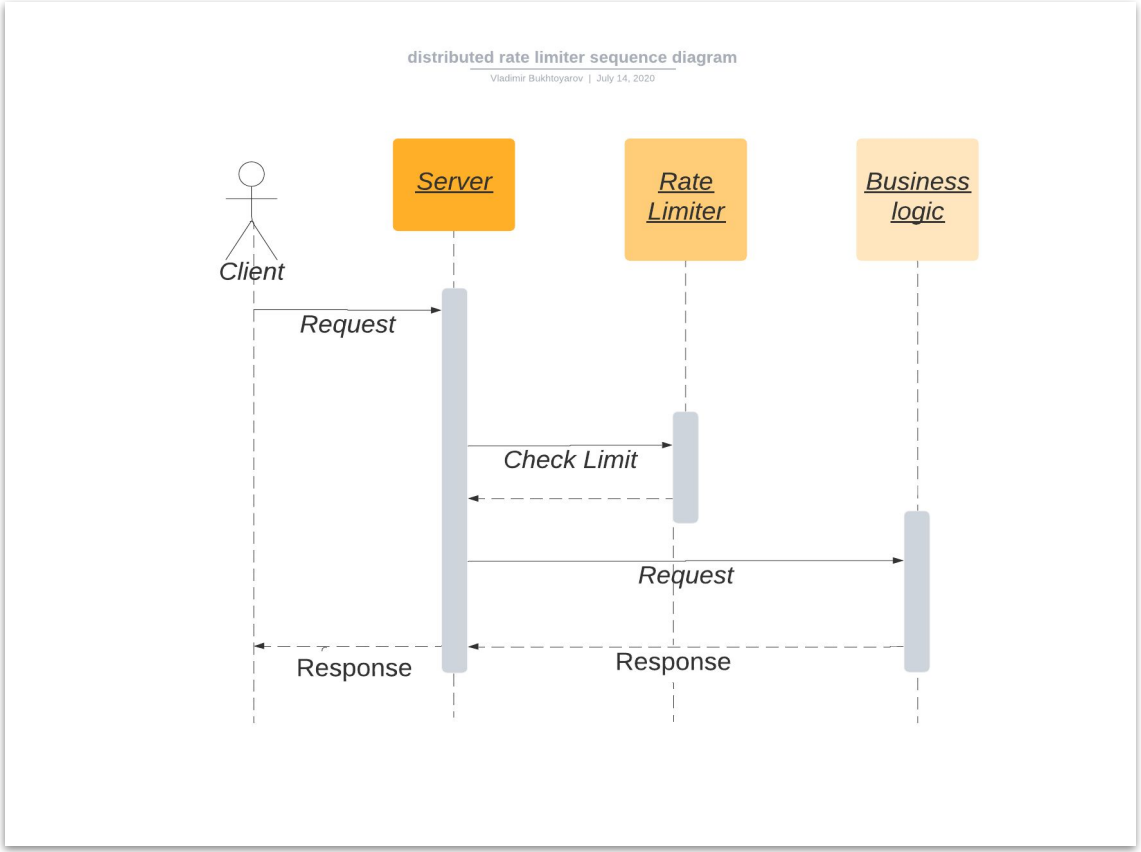
```java
@Override
public boolean tryAcquire(int permits) {
    State newState = new State();
    while (true) {
        long now = System.nanoTime();
        State previousState = stateReference.get();
        newState.availableTokens = previousState.availableTokens;
        newState.lastRefillNanotime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

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

# Реализуем распределенный Token Bucket

- Выбор технологии.
- Пример кода.
- Обзор проблем.

# Проблема RateLimiter в кластере:



# Требования к RateLimiter в кластере:

- Работать быстро. Желательно укладываться в 1 миллисекунду.
- Корректно работать в условиях параллельного доступа.
- Решение должно основываться на известных проверенных инструментах.

# Пути реализации RateLimiter в кластере:

- DBMS + Select for Update?
  Always 3 network hops.
- MemCached, MongoDB,... + Compare And SWAP???
  2 network hops in lucky case, but **unpredictable** in general.
- DBMS + Stored procedures?!
  Always 1 network hops.
- Redis, Tarantool, Aerospike + Stored procedures!
  Always 1 network hops.
- JSR 107(Hazelcast, Coherence, Ignite) + EntryProcessor!!!
  Always 1 network hops.

# 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 remote part:

```
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 remote part:

```
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 remote part:

```
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 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;
}
```

# Итоги:

- Научились имплементировать Lock-Free Token Bucket.
- Научились избегать арифметические ловушки.
- Научились имплементировать distributed Token Bucket с помощью Apache Ignite.

# Thank you

**Vladimir Bukhtoyarov**

**jsecoder@mail.ru**

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