UNIVERSITÀ
DEGLI STUDI
DI PADOVA

CONCURRENT AND REAL TIME PROGRAMMING

Lecturers:
Prof. Gabriele Manduchi
Prof. Andrea Rigoni Garola

- 20/2/2024 -

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Topic 4: Producer - Consumer Flow Control

Student: Nguyen Tho The Cuong – 2106235

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

OUTLINE

1. Introduction and System Design
2. Flow Control Algorithm
3. Metrics Monitoring
4. Results
5. Conclusion

01

Introduction and System Design

UNIVERSITÀ DEGLI STUDI DI PADOVA

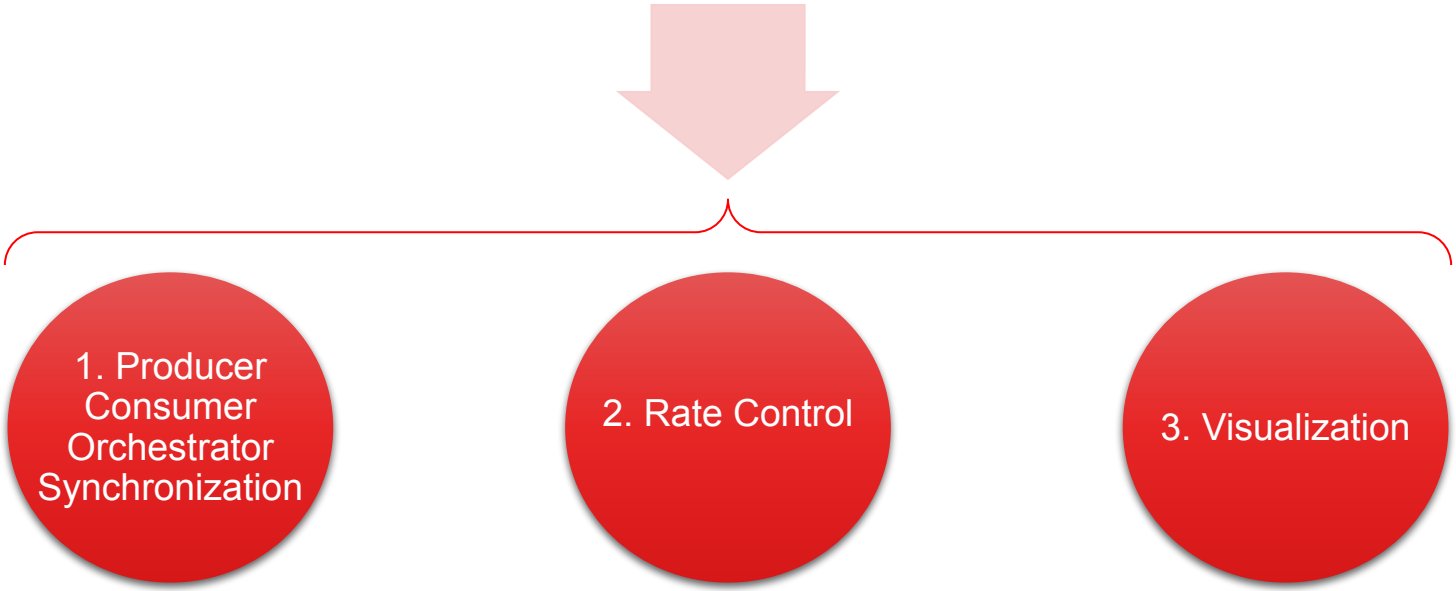
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Introduction and System Design

4. Producer - (single) consumer program with dynamic message rate adjustment. The consumer shall consume messages at a given rate, that is, with a given delay simulating the consumed message usage. An actor (task or process) separate from producer and consumer shall periodically check the message queue length and if the length is below a given threshold, it will increase the production rate. Otherwise (i.e. the message length is above the given threshold), it will decrease the production rate.

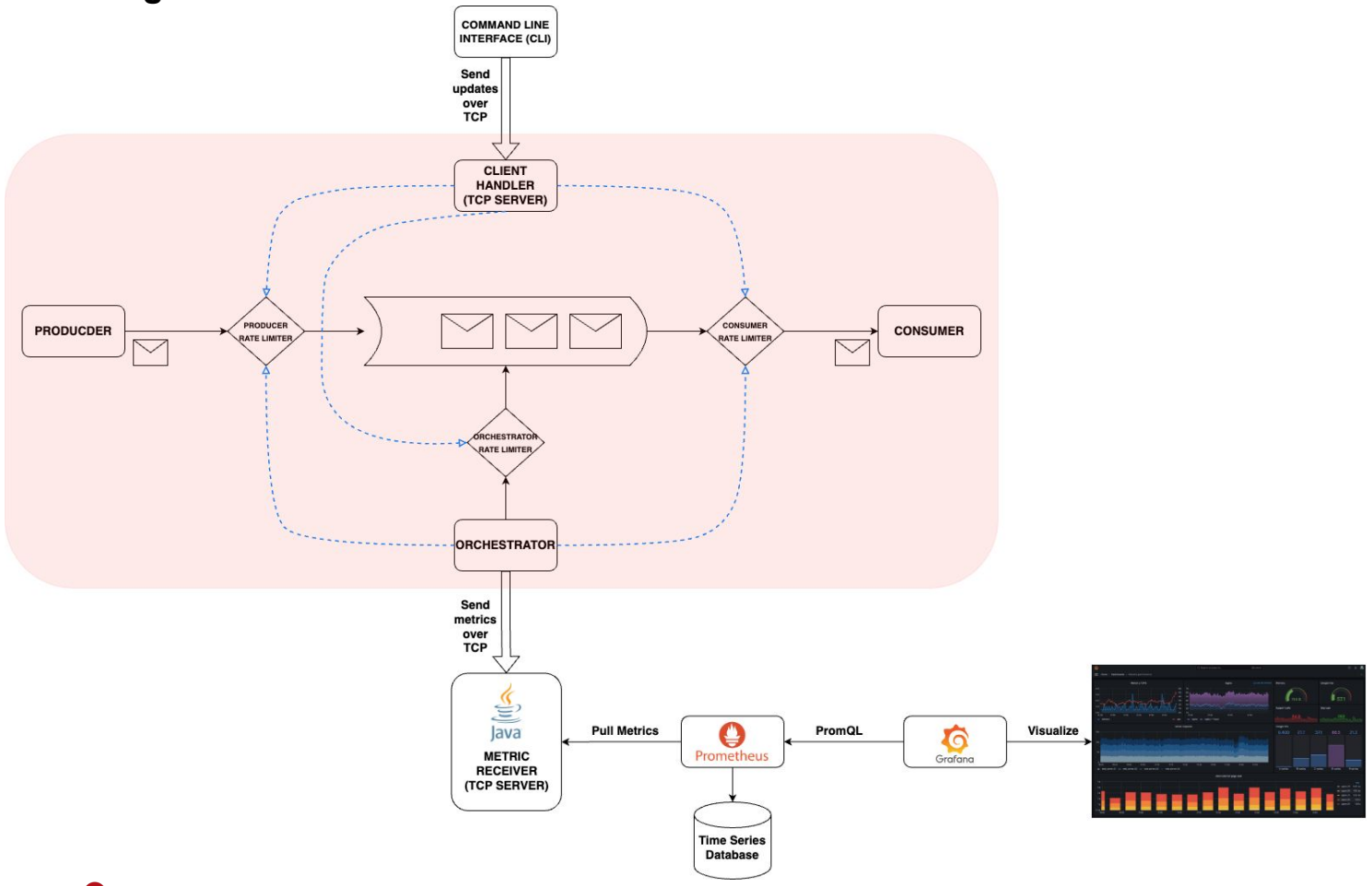
Introduction and System Design

4. Producer - (single) consumer program with dynamic message rate adjustment. The consumer shall consume messages at a given rate, that is, with a given delay simulating the consumed message usage. An actor (task or process) separate from producer and consumer shall periodically check the message queue length and if the length is below a given threshold, it will increase the production rate. Otherwise (i.e. the message length is above the given threshold), it will decrease the production rate.



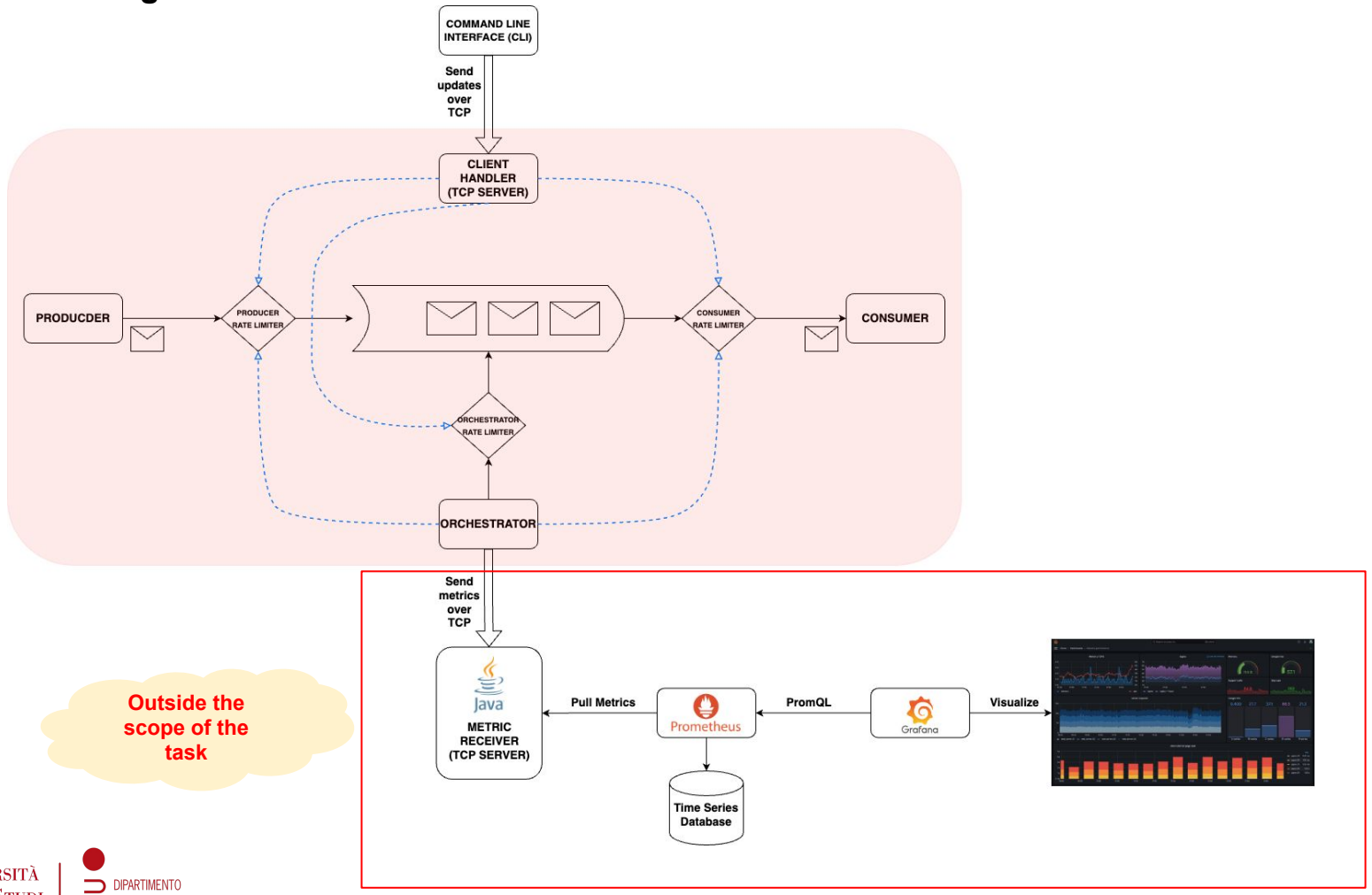
Introduction and System Design

❖ Full Design



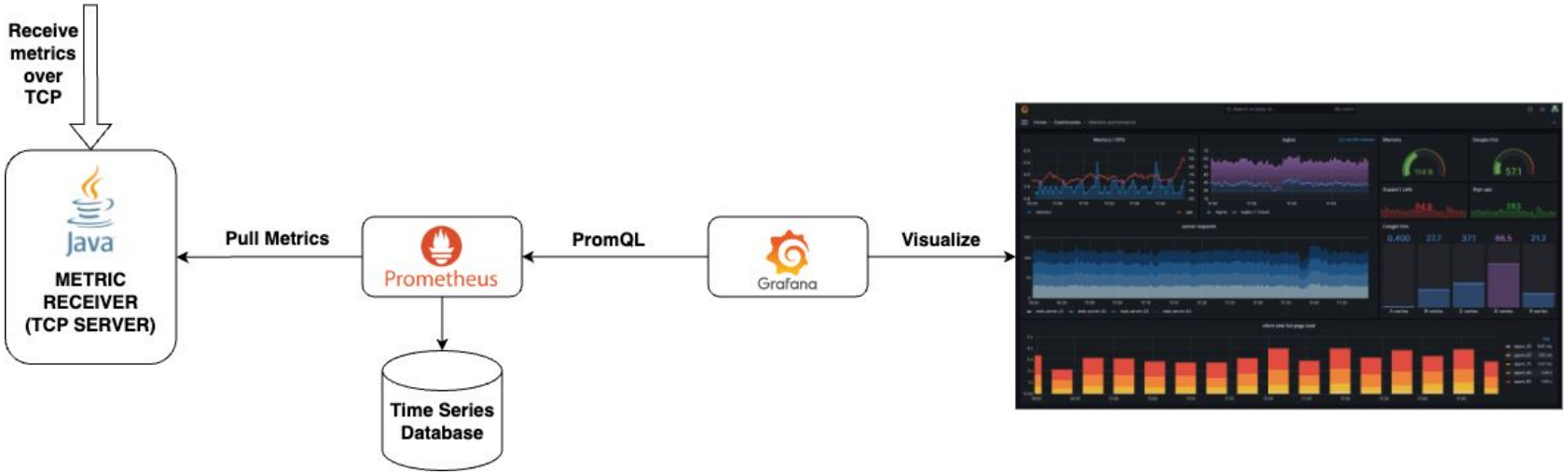
Introduction and System Design

❖ Full Design



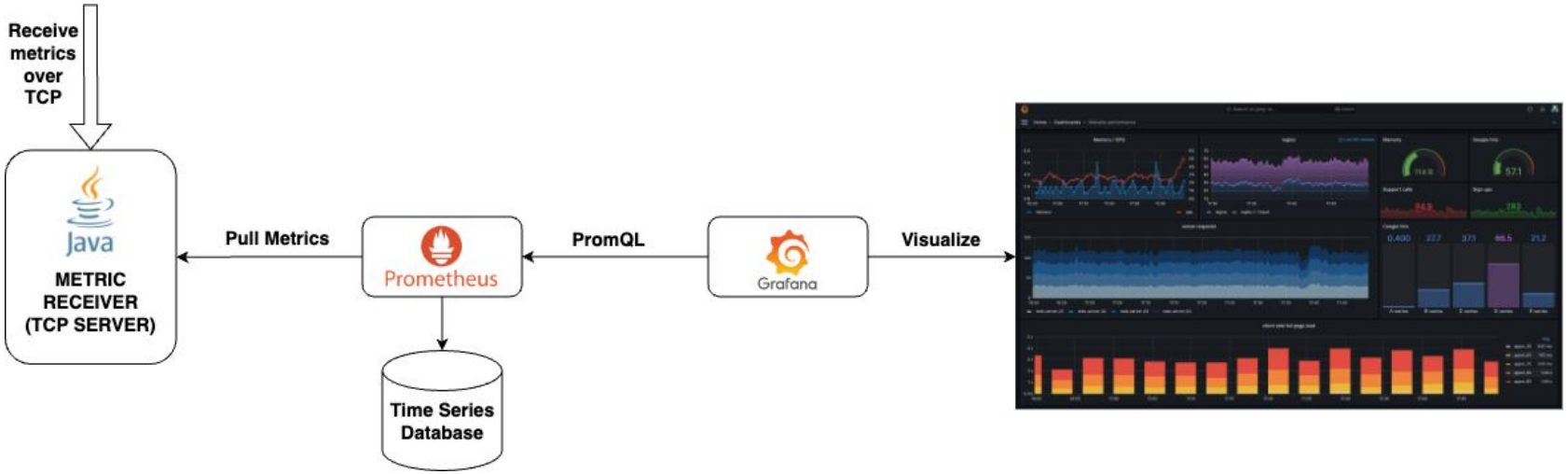
Introduction and System Design

- **Outside the scope of the task:**



Introduction and System Design

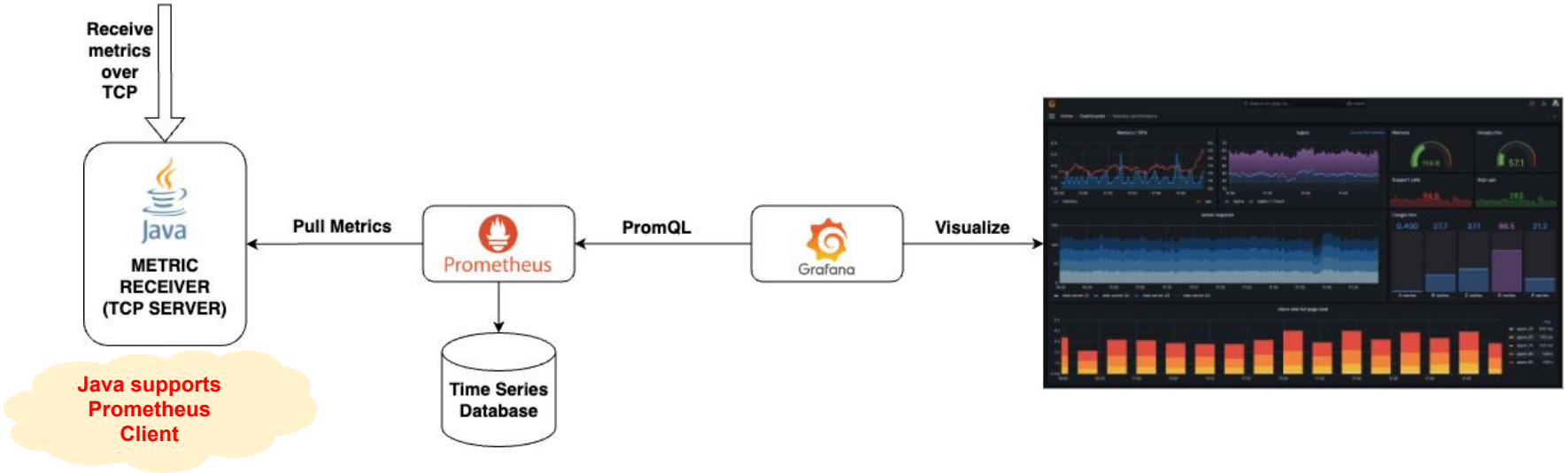
- **Outside the scope of the task:**



3. Visualization

Introduction and System Design

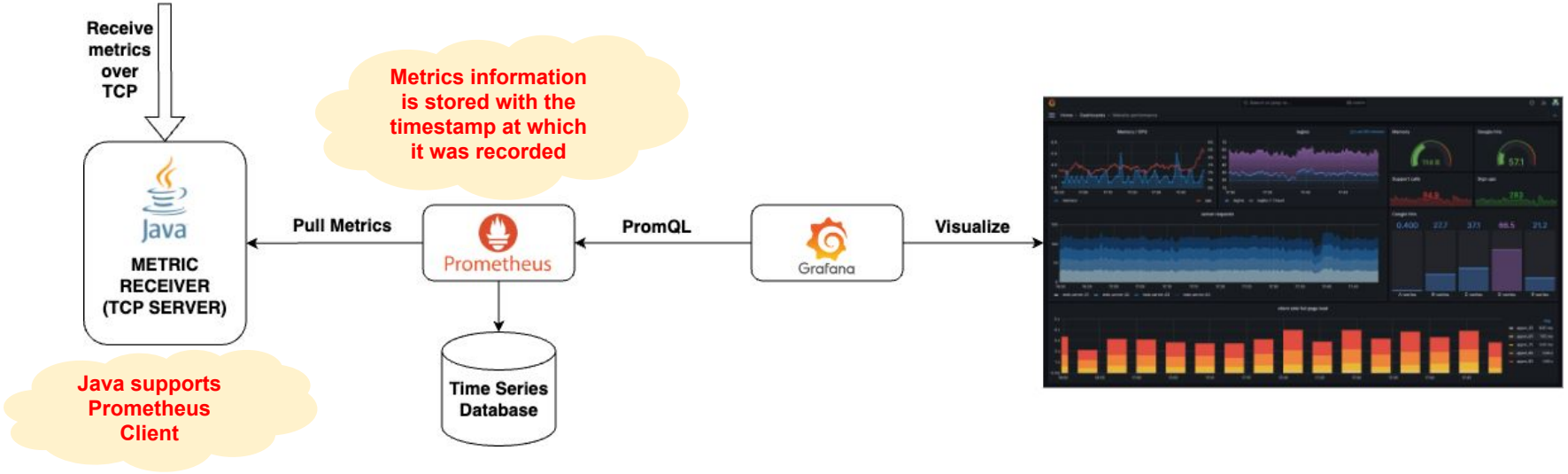
- **Outside the scope of the task:**



3. Visualization

Introduction and System Design

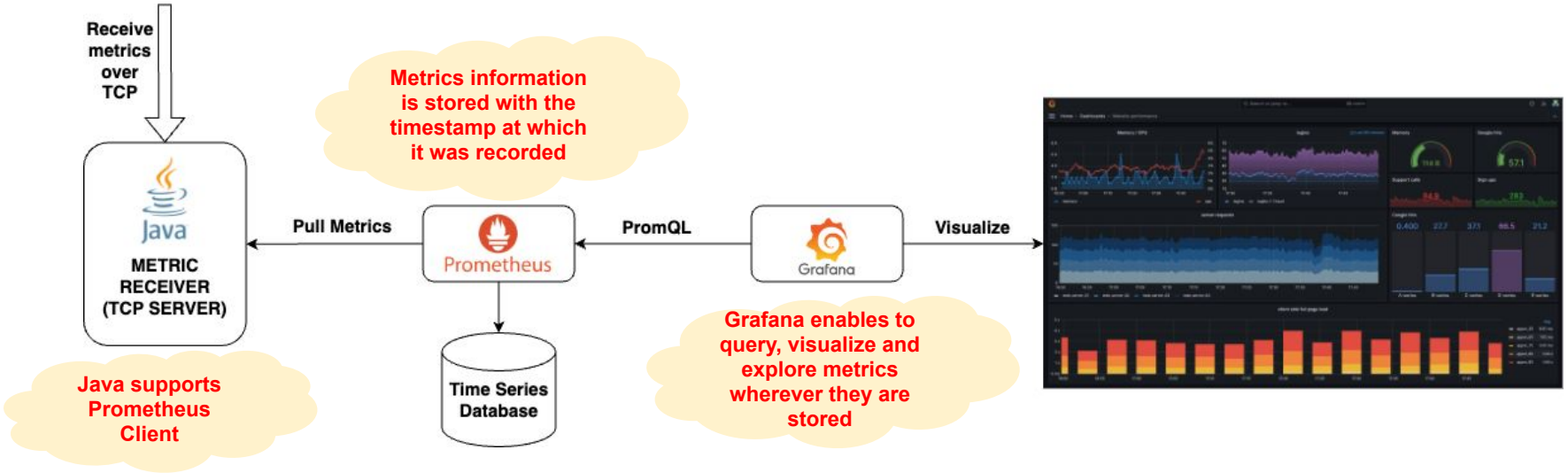
- **Outside the scope of the task:**



3. Visualization

Introduction and System Design

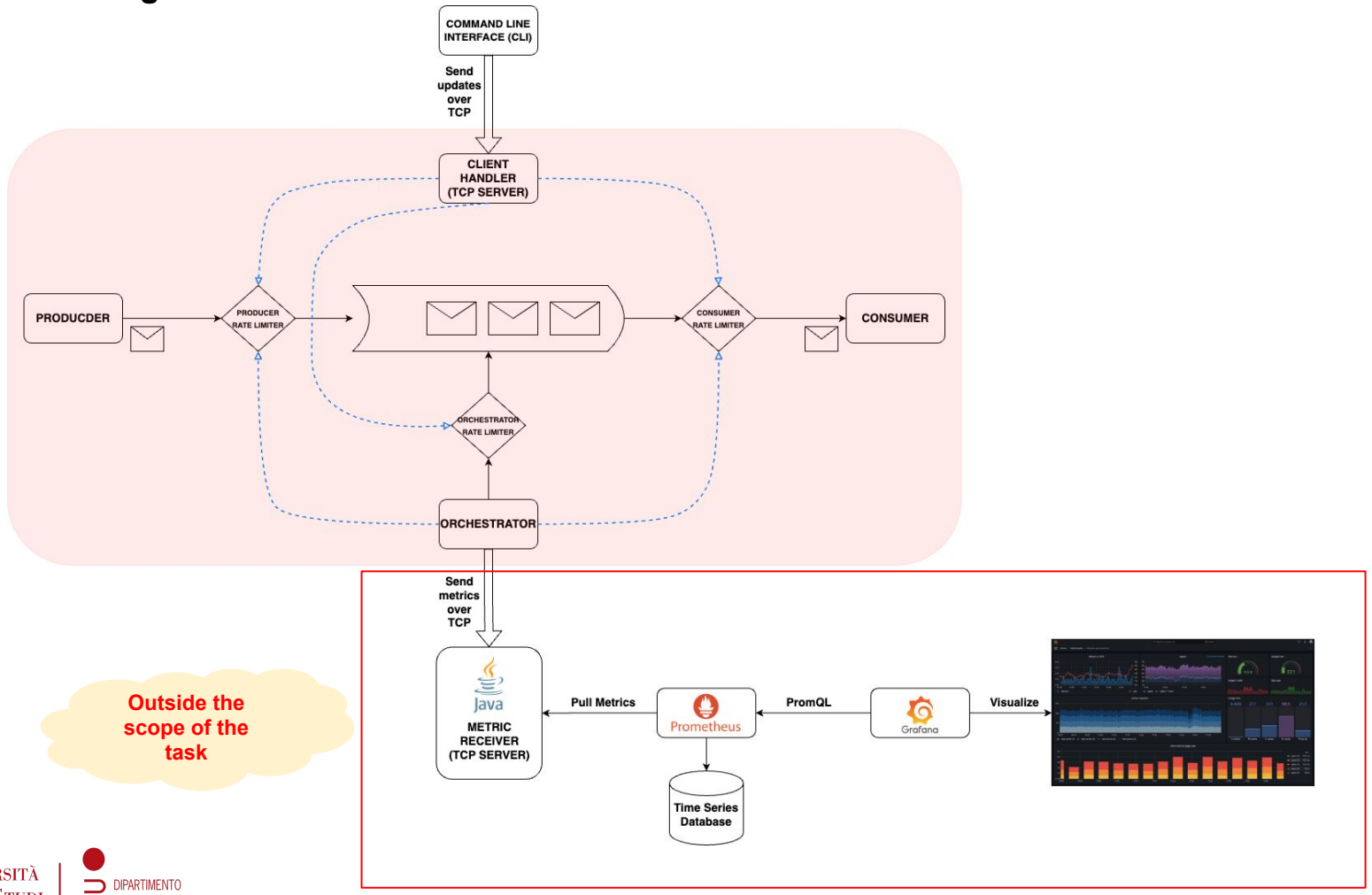
- **Outside the scope of the task:**



3. Visualization

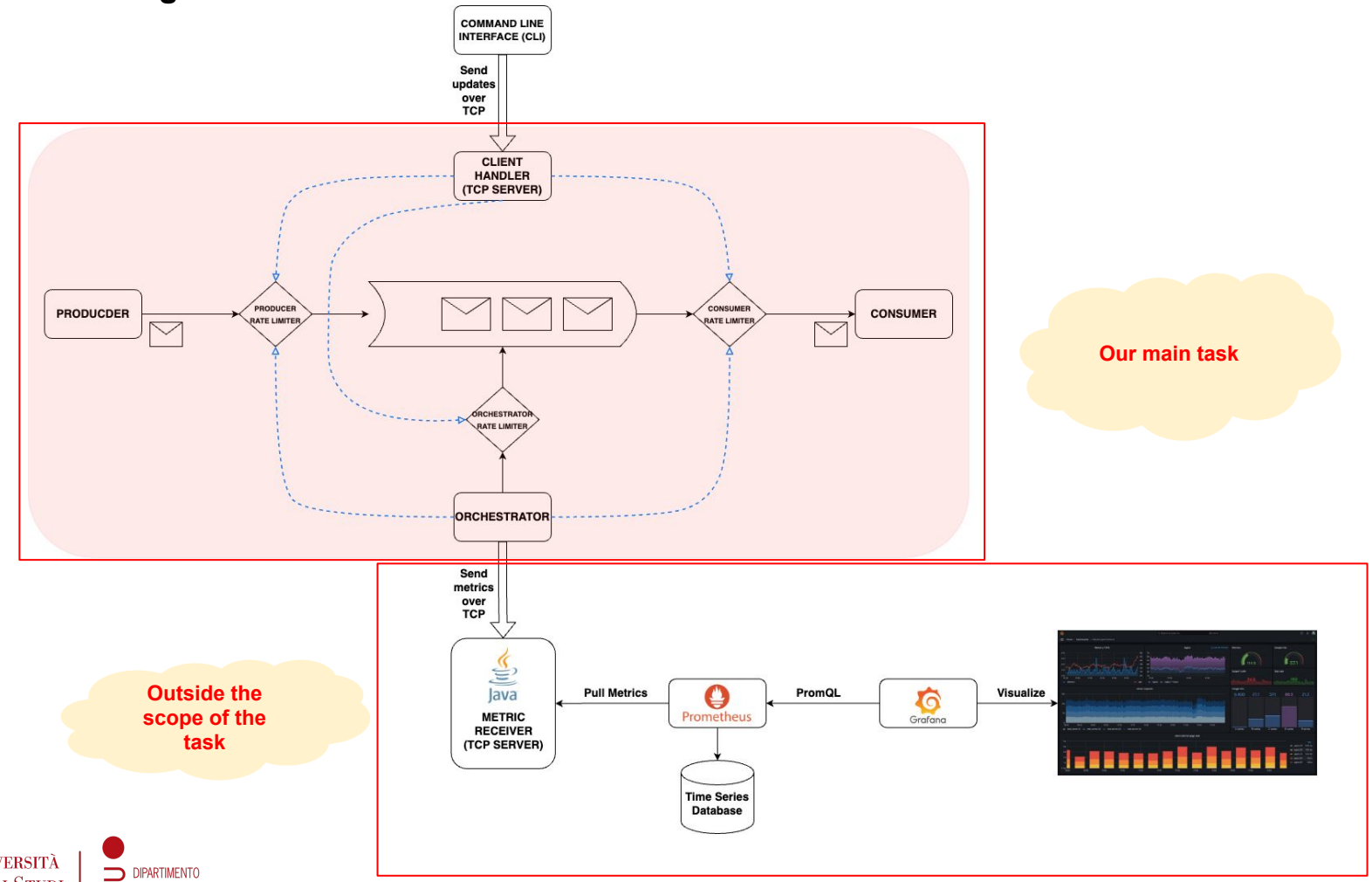
Introduction and System Design

❖ Full Design



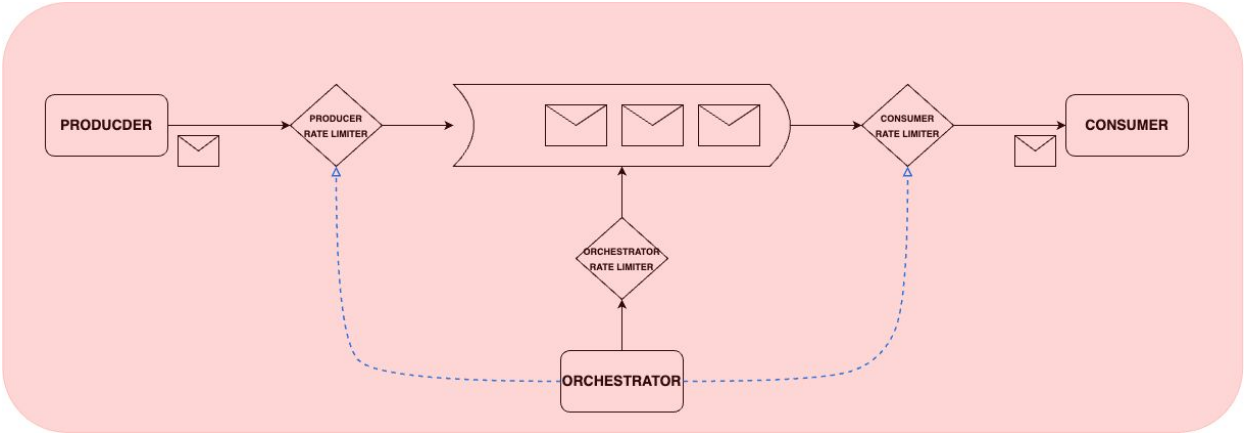
Introduction and System Design

❖ Full Design



Introduction and System Design

- **Solution for the task:**

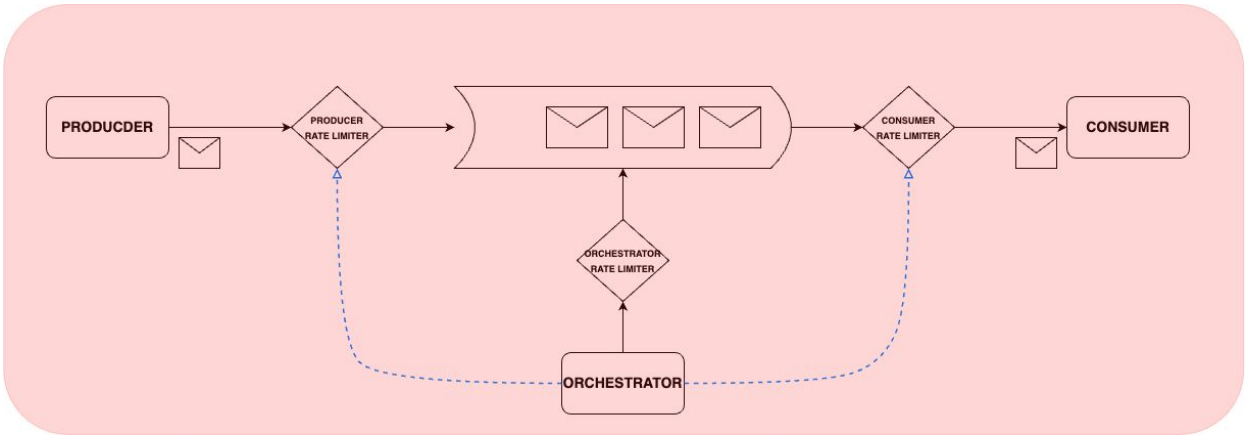


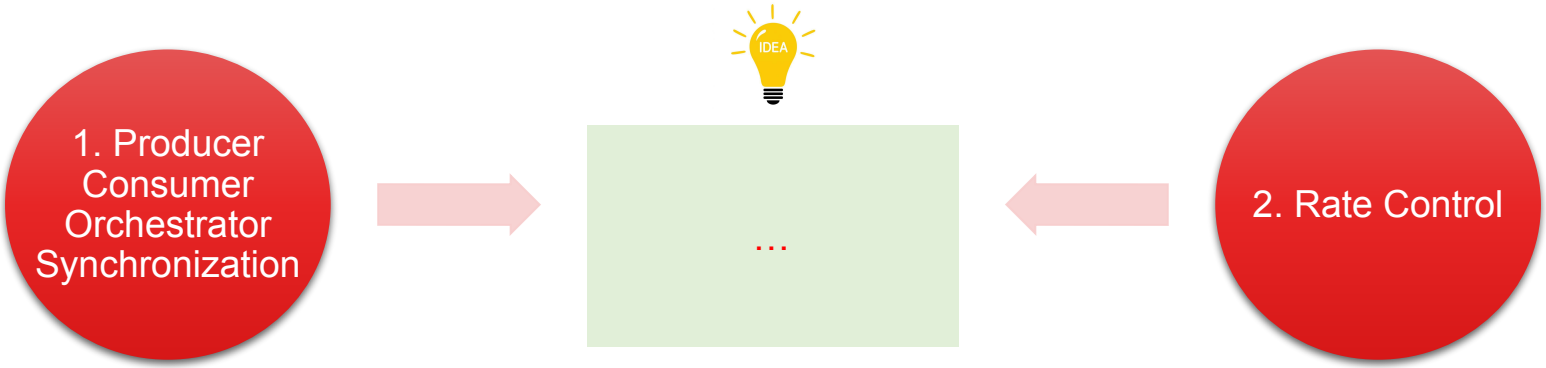
1. Producer Consumer Orchestrator Synchronization

2. Rate Control

Introduction and System Design

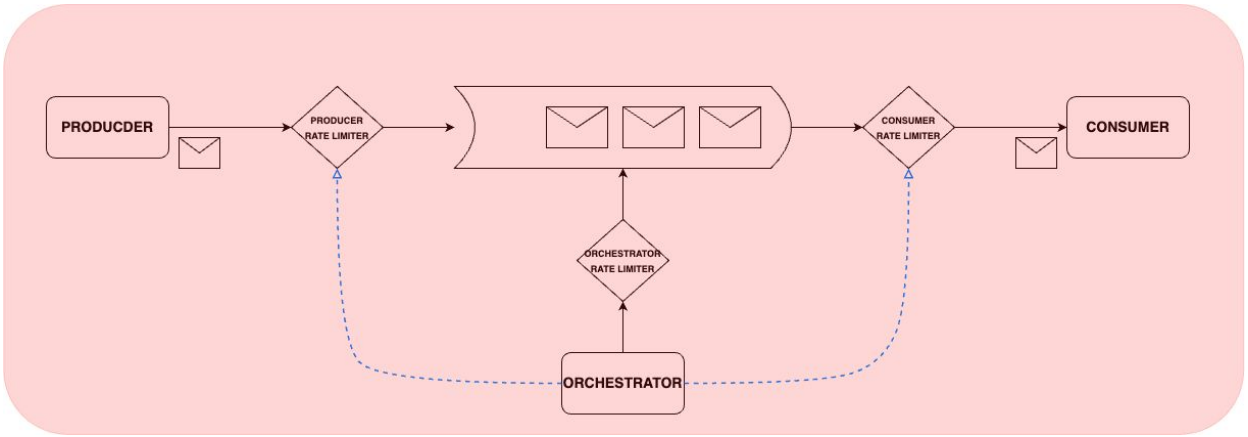
- **Solution for the task:**





Introduction and System Design

- **Solution for the task:**



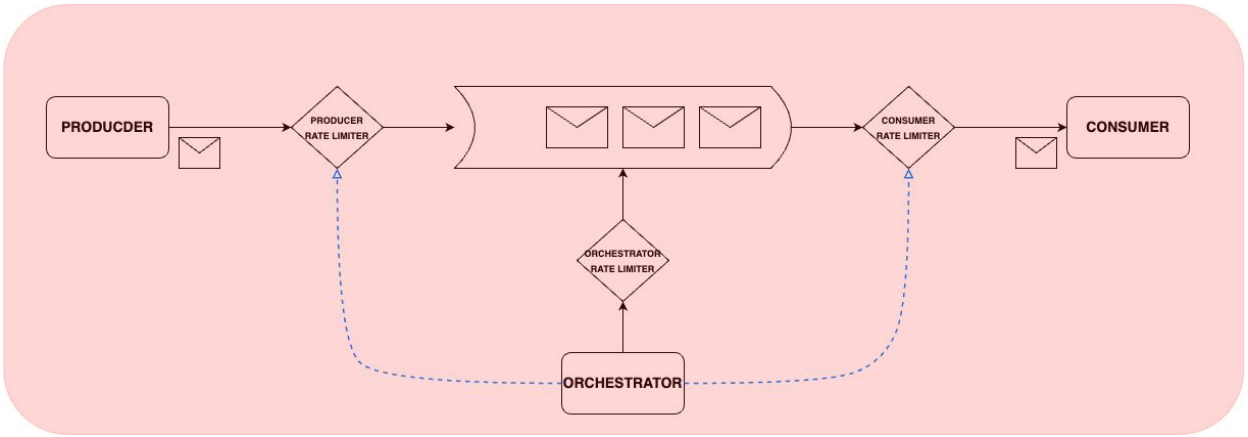
1. Producer Consumer Orchestrator Synchronization

- ✔ Semaphore
- ✔ 3-Thread model
- ✔ Rate Limiter

2. Rate Control

Introduction and System Design

- **Solution for the task:**



1. Producer Consumer Orchestrator Synchronization

- ✔ Semaphore
- ✔ 3-Thread model
- ✔ Rate Limiter

Token Bucket Algorithm

2. Rate Control

02

Flow Control Algorithm

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Flow Control Algorithm

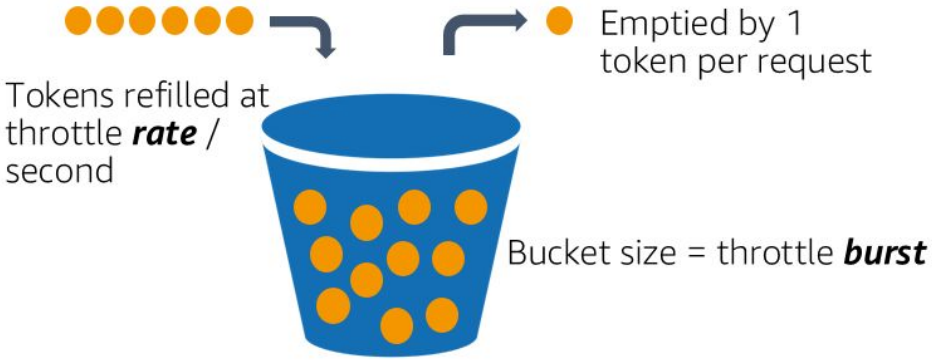
- Idea: Token Bucket Algorithm

- ➔ Input: Assign a limiting rate
- ➔ Output: Allow a number of permits per one unit of time (e.g. per second)

Flow Control Algorithm

- **Idea: Token Bucket Algorithm**

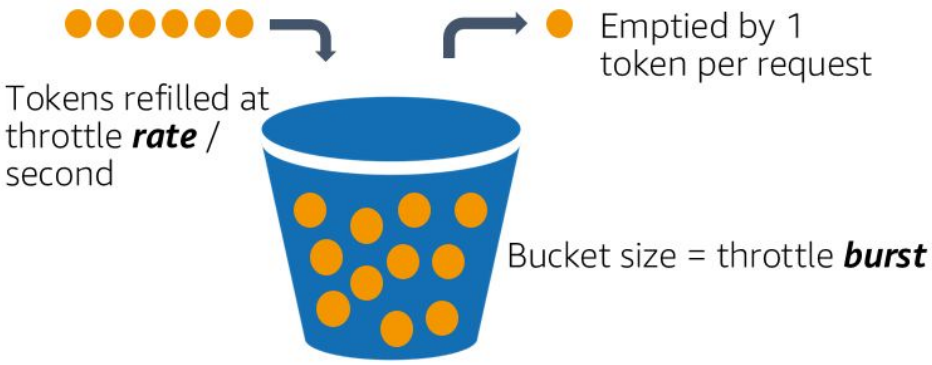
- **Input: Assign a limiting rate**
- **Output: Allow a number of permits per one unit of time (e.g. per second)**

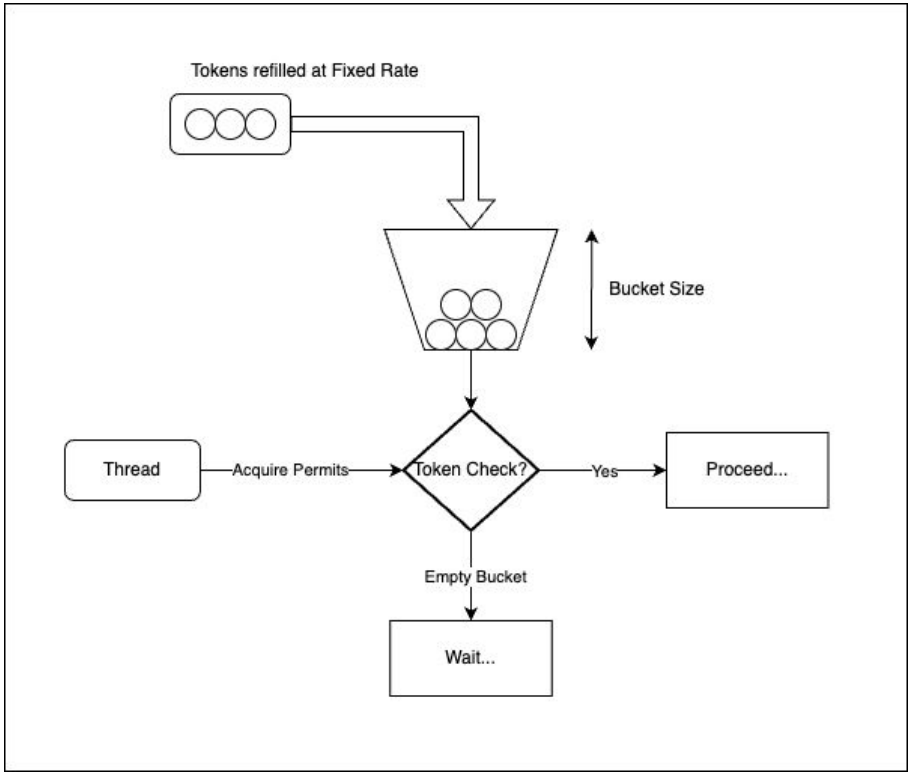


Flow Control Algorithm

- **Idea: Token Bucket Algorithm**

- **Input: Assign a limiting rate**
- **Output: Allow a number of permits per one unit of time (e.g. per second)**





Flow Control Algorithm

❖ Implementation and Usage of Rate Limiter

```c
typedef struct {
    // thread-safe
    pthread_mutex_t mutex;
    double interval;
    usec_t next_free;

} RateLimiter;

/* Claim next permits */
usec_t claim_next(RateLimiter *rate_limiter, int permits);

/* Claim next permits and then usleep if not permited */
usec_t acquire_permits(RateLimiter *rate_limiter, int permits);

/* Acquire only one permit and then usleep if not permited */
usec_t acquire(RateLimiter *rate_limiter);

/* Change the limiting rate */
void set_rate(RateLimiter *rate_limiter, double rate);

double get_rate(RateLimiter *rate_limiter);
RateLimiter *get_rate_limiter(double rate);
```

```c
/* Producer code. Passed argument is not used */
static void *producer(void *arg)
{
    int item = 0;
    while (1)
    {
        /* Wait for availability of at least one empty slot */
        sem_wait(room_vailable_sem);

        /* Limit rate */
        acquire(prod_rate_limiter);

        /* Enter critical section */
        sem_wait(mutex_sem);

        /* Write data item */
        buffer[writeIdx] = item;
        /* Update write index */
        writeIdx = (writeIdx + 1) % BUFFER_SIZE;

        /* Update metrics */
        prod_count += 1;
        queue_size += 1;

        /* Signal that a new data slot is available */
        sem_post(data_available_sem);

        /* Exit critical section */
        sem_post(mutex_sem);

        /* Produce data item and take actions (e.g return) */
    }
}
```

Flow Control Algorithm

❖ Implementation and Usage of Rate Limiter

```c
typedef struct {
    // thread-safe
    pthread_mutex_t mutex;
    double interval;
    usec_t next_free;

} RateLimiter;

/* Claim next permits */
usec_t claim_next(RateLimiter *rate_limiter, int permits);

/* Claim next permits and then usleep if not permited */
usec_t acquire_permits(RateLimiter *rate_limiter, int permits);

/* Acquire only one permit and then usleep if not permited */
usec_t acquire(RateLimiter *rate_limiter);

/* Change the limiting rate */
void set_rate(RateLimiter *rate_limiter, double rate);

double get_rate(RateLimiter *rate_limiter);
RateLimiter *get_rate_limiter(double rate);
```

```c
/* Producer code. Passed argument is not used */
static void *producer(void *arg)
{
    int item = 0;
    while (1)
    {
        /* Wait for availability of at least one empty slot */
        sem_wait(room_vailable_sem);

        /* Limit rate */
        acquire(prod_rate_limiter);

        /* Enter critical section */
        sem_wait(mutex_sem);

        /* Write data item */
        buffer[writeIdx] = item;
        /* Update write index */
        writeIdx = (writeIdx + 1) % BUFFER_SIZE;

        /* Update metrics */
        prod_count += 1;
        queue_size += 1;

        /* Signal that a new data slot is available */
        sem_post(data_available_sem);

        /* Exit critical section */
        sem_post(mutex_sem);

        /* Produce data item and take actions (e.g return) */
    }
}
```

Flow Control Algorithm

❖ Implementation and Usage of Rate Limiter

```c
typedef struct {
    // thread-safe
    pthread_mutex_t mutex;
    double interval;
    usec_t next_free;

} RateLimiter;

/* Claim next permits */
usec_t claim_next(RateLimiter *rate_limiter, int permits);

/* Claim next permits and then usleep if not permited */
usec_t acquire_permits(RateLimiter *rate_limiter, int permits);

/* Acquire only one permit and then usleep if not permited */
usec_t acquire(RateLimiter *rate_limiter);

/* Change the limiting rate */
void set_rate(RateLimiter *rate_limiter, double rate);

double get_rate(RateLimiter *rate_limiter);
RateLimiter *get_rate_limiter(double rate);
```

```c
/* Producer code. Passed argument is not used */
static void *producer(void *arg)
{
    int item = 0;
    while (1)
    {
        /* Wait for availability of at least one empty slot */
        sem_wait(room_vailable_sem);

        /* Limit rate */
        acquire(prod_rate_limiter);

        /* Enter critical section */
        sem_wait(mutex_sem);

        /* Write data item */
        buffer[writeIdx] = item;
        /* Update write index */
        writeIdx = (writeIdx + 1) % BUFFER_SIZE;

        /* Update metrics */
        prod_count += 1;
        queue_size += 1;

        /* Signal that a new data slot is available */
        sem_post(data_available_sem);

        /* Exit critical section */
        sem_post(mutex_sem);

        /* Produce data item and take actions (e.g return) */
    }
}
```

Flow Control Algorithm

❖ Implementation and Usage of Rate Limiter

```
typedef struct {
    // thread-safe
    pthread_mutex_t mutex;
    double interval;
    usec_t next_free;

} RateLimiter;

/* Claim next permits */
usec_t claim_next(RateLimiter *rate_limiter, int permits);

/* Claim next permits and then usleep if not permited */
usec_t acquire_permits(RateLimiter *rate_limiter, int permits);

/* Acquire only one permit and then usleep if not permited */
usec_t acquire(RateLimiter *rate_limiter);

/* Change the limiting rate */
void set_rate(RateLimiter *rate_limiter, double rate);

double get_rate(RateLimiter *rate_limiter);
RateLimiter *get_rate_limiter(double rate);
```

```
/* Producer code. Passed argument is not used */
static void *producer(void *arg)
{
    int item = 0;
    while (1)
    {
        /* Wait for availability of at least one empty slot */
        sem_wait(room_vailable_sem);

        /* Limit rate */
        acquire(prod_rate_limiter);

        /* Enter critical section */
        sem_wait(mutex_sem);

        /* Write data item */
        buffer[writeIdx] = item;
        /* Update write index */
        writeIdx = (writeIdx + 1) % BUFFER_SIZE;

        /* Update metrics */
        prod_count += 1;
        queue_size += 1;

        /* Signal that a new data slot is available */
        sem_post(data_available_sem);

        /* Exit critical section */
        sem_post(mutex_sem);

        /* Produce data item and take actions (e.g return) */
    }
}
```

Blocked if not permitted

Flow Control Algorithm

❖ Implementation and Usage of Rate Limiter

```c
typedef struct {
    // thread-safe
    pthread_mutex_t mutex;
    double interval;
    usec_t next_free;

} RateLimiter;

/* Claim next permits */
usec_t claim_next(RateLimiter *rate_limiter, int permits);

/* Claim next permits and then usleep if not permited */
usec_t acquire_permits(RateLimiter *rate_limiter, int permits);

/* Acquire only one permit and then usleep if not permited */
usec_t acquire(RateLimiter *rate_limiter);

/* Change the limiting rate */
void set_rate(RateLimiter *rate_limiter, double rate);

double get_rate(RateLimiter *rate_limiter);
RateLimiter *get_rate_limiter(double rate);
```

```c
/* Producer code. Passed argument is not used */
static void *producer(void *arg)
{
    int item = 0;
    while (1)
    {
        /* Wait for availability of at least one empty slot */
        sem_wait(room_vailable_sem);

        /* Limit rate */
        acquire(prod_rate_limiter);

        /* Enter critical section */
        sem_wait(mutex_sem);

        /* Write data item */
        buffer[writeIdx] = item;
        /* Update write index */
        writeIdx = (writeIdx + 1) % BUFFER_SIZE;

        /* Update metrics */
        prod_count += 1;
        queue_size += 1;

        /* Signal that a new data slot is available */
        sem_post(data_available_sem);

        /* Exit critical section */
        sem_post(mutex_sem);

        /* Produce data item and take actions (e.g return) */
    }
}
```

Blocked if not permitted

Flow Control Algorithm

❖ Implementation and Usage of Rate Limiter

```c
typedef struct {
    // thread-safe
    pthread_mutex_t mutex;
    double interval;
    usec_t next_free;

} RateLimiter;

/* Claim next permits */
usec_t claim_next(RateLimiter *rate_limiter, int permits);

/* Claim next permits and then usleep if not permited */
usec_t acquire_permits(RateLimiter *rate_limiter, int permits);

/* Acquire only one permit and then usleep if not permited */
usec_t acquire(RateLimiter *rate_limiter);

/* Change the limiting rate */
void set_rate(RateLimiter *rate_limiter, double rate);

double get_rate(RateLimiter *rate_limiter);
RateLimiter *get_rate_limiter(double rate);
```

```c
/* Producer code. Passed argument is not used */
static void *producer(void *arg)
{
    int item = 0;
    while (1)
    {
        /* Wait for availability of at least one empty slot */
        sem_wait(room_vailable_sem);

        /* Limit rate */
        acquire(prod_rate_limiter);

        /* Enter critical section */
        sem_wait(mutex_sem);

        /* Write data item */
        buffer[writeIdx] = item;
        /* Update write index */
        writeIdx = (writeIdx + 1) % BUFFER_SIZE;

        /* Update metrics */
        prod_count += 1;
        queue_size += 1;

        /* Signal that a new data slot is available */
        sem_post(data_available_sem);

        /* Exit critical section */
        sem_post(mutex_sem);

        /* Produce data item and take actions (e.g return) */
    }
}
```

Blocked if not permitted

Update accumulated count of enqueued items and the current queue size

Flow Control Algorithm

- **Regulate the rate of Producer**

```
static void control_flow(int cur_queue_size) {
    double cur_prod_rate = get_rate(prod_rate_limiter);
    double cur_cons_rate = get_rate(cons_rate_limiter);
    double updated_prod_rate = cur_prod_rate;

    if (cur_queue_size > queue_size_threshold) {
        // decreasing rate of producer
        updated_prod_rate -= rate_change_step; // rate_change_step = 0.5

        // min rate = cur_cons_rate / 2
        if (updated_prod_rate < cur_cons_rate / 2) {
            updated_prod_rate = cur_cons_rate / 2;
        }
    } else {
        // increasing rate of producer
        updated_prod_rate += rate_change_step; // rate_change_step = 0.5

        // max rate = 2 * cons_rate
        if (updated_prod_rate > cur_cons_rate * 2) {
            updated_prod_rate = cur_cons_rate * 2;
        }
    }

    // update rate of producer
    set_rate(prod_rate_limiter, updated_prod_rate);
}
```

Flow Control Algorithm

- **Regulate the rate of Producer**

```
static void control_flow(int cur_queue_size) {
    double cur_prod_rate = get_rate(prod_rate_limiter);
    double cur_cons_rate = get_rate(cons_rate_limiter);
    double updated_prod_rate = cur_prod_rate;

    if (cur_queue_size > queue_size_threshold) {
        // decreasing rate of producer
        updated_prod_rate -= rate_change_step; // rate_change_step = 0.5

        // min rate = cur_cons_rate / 2
        if (updated_prod_rate < cur_cons_rate / 2) {
            updated_prod_rate = cur_cons_rate / 2;
        }
    } else {
        // increasing rate of producer
        updated_prod_rate += rate_change_step; // rate_change_step = 0.5

        // max rate = 2 * cons_rate
        if (updated_prod_rate > cur_cons_rate * 2) {
            updated_prod_rate = cur_cons_rate * 2;
        }
    }

    // update rate of producer
    set_rate(prod_rate_limiter, updated_prod_rate);
}
```

Decrease rate

Flow Control Algorithm

- **Regulate the rate of Producer**

```
static void control_flow(int cur_queue_size) {
    double cur_prod_rate = get_rate(prod_rate_limiter);
    double cur_cons_rate = get_rate(cons_rate_limiter);
    double updated_prod_rate = cur_prod_rate;

    if (cur_queue_size > queue_size_threshold) {
        // decreasing rate of producer
        updated_prod_rate -= rate_change_step; // rate_change_step = 0.5

        // min rate = cur_cons_rate / 2
        if (updated_prod_rate < cur_cons_rate / 2) {
            updated_prod_rate = cur_cons_rate / 2;
        }
    } else {
        // increasing rate of producer
        updated_prod_rate += rate_change_step; // rate_change_step = 0.5

        // max rate = 2 * cons_rate
        if (updated_prod_rate > cur_cons_rate * 2) {
            updated_prod_rate = cur_cons_rate * 2;
        }
    }

    // update rate of producer
    set_rate(prod_rate_limiter, updated_prod_rate);
}
```

Decrease rate

Increase rate

Flow Control Algorithm

- **Regulate the rate of Producer**

```
static void control_flow(int cur_queue_size) {
    double cur_prod_rate = get_rate(prod_rate_limiter);
    double cur_cons_rate = get_rate(cons_rate_limiter);
    double updated_prod_rate = cur_prod_rate;

    if (cur_queue_size > queue_size_threshold) {
        // decreasing rate of producer
        updated_prod_rate -= rate_change_step; // rate_change_step = 0.5

        // min rate = cur_cons_rate / 2
        if (updated_prod_rate < cur_cons_rate / 2) {
            updated_prod_rate = cur_cons_rate / 2;
        }
    } else {
        // increasing rate of producer
        updated_prod_rate += rate_change_step; // rate_change_step = 0.5

        // max rate = 2 * cons_rate
        if (updated_prod_rate > cur_cons_rate * 2) {
            updated_prod_rate = cur_cons_rate * 2;
        }
    }

    // update rate of producer
    set_rate(prod_rate_limiter, updated_prod_rate);
}
```

Decrease rate

Increase rate

Apply new rate

03

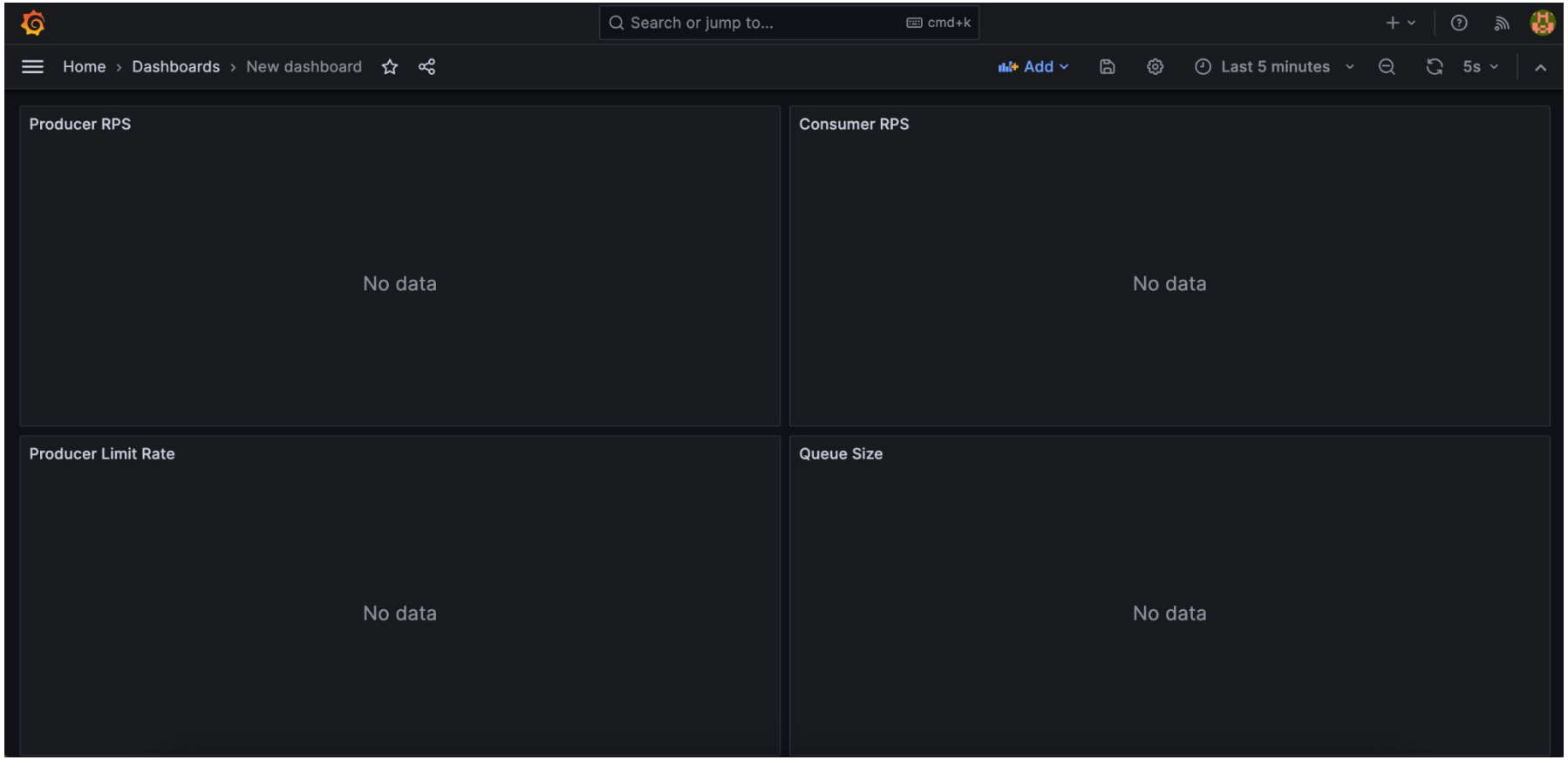
Metrics Monitoring

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

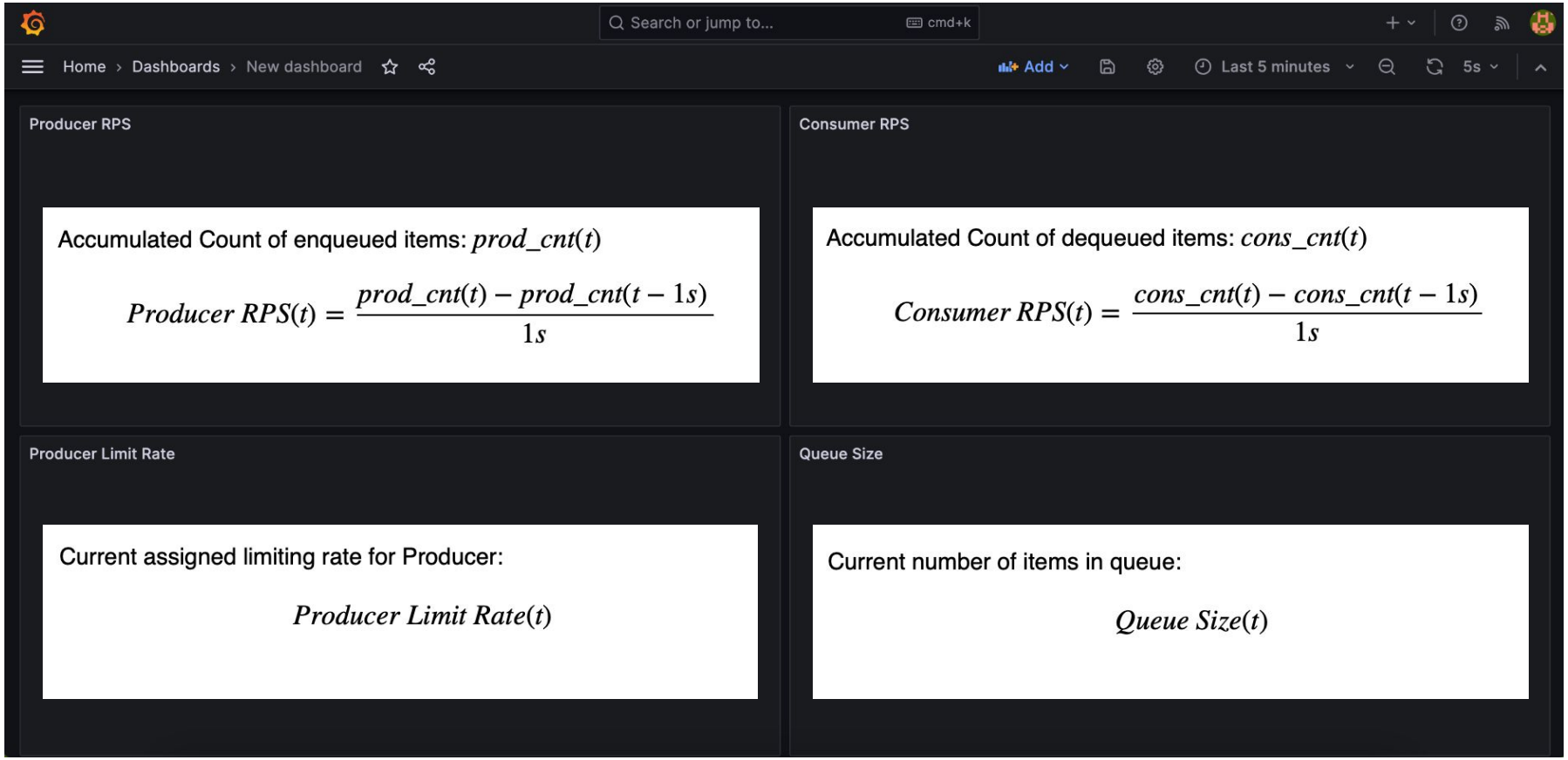
Metrics Monitoring

- **At the instant *t***



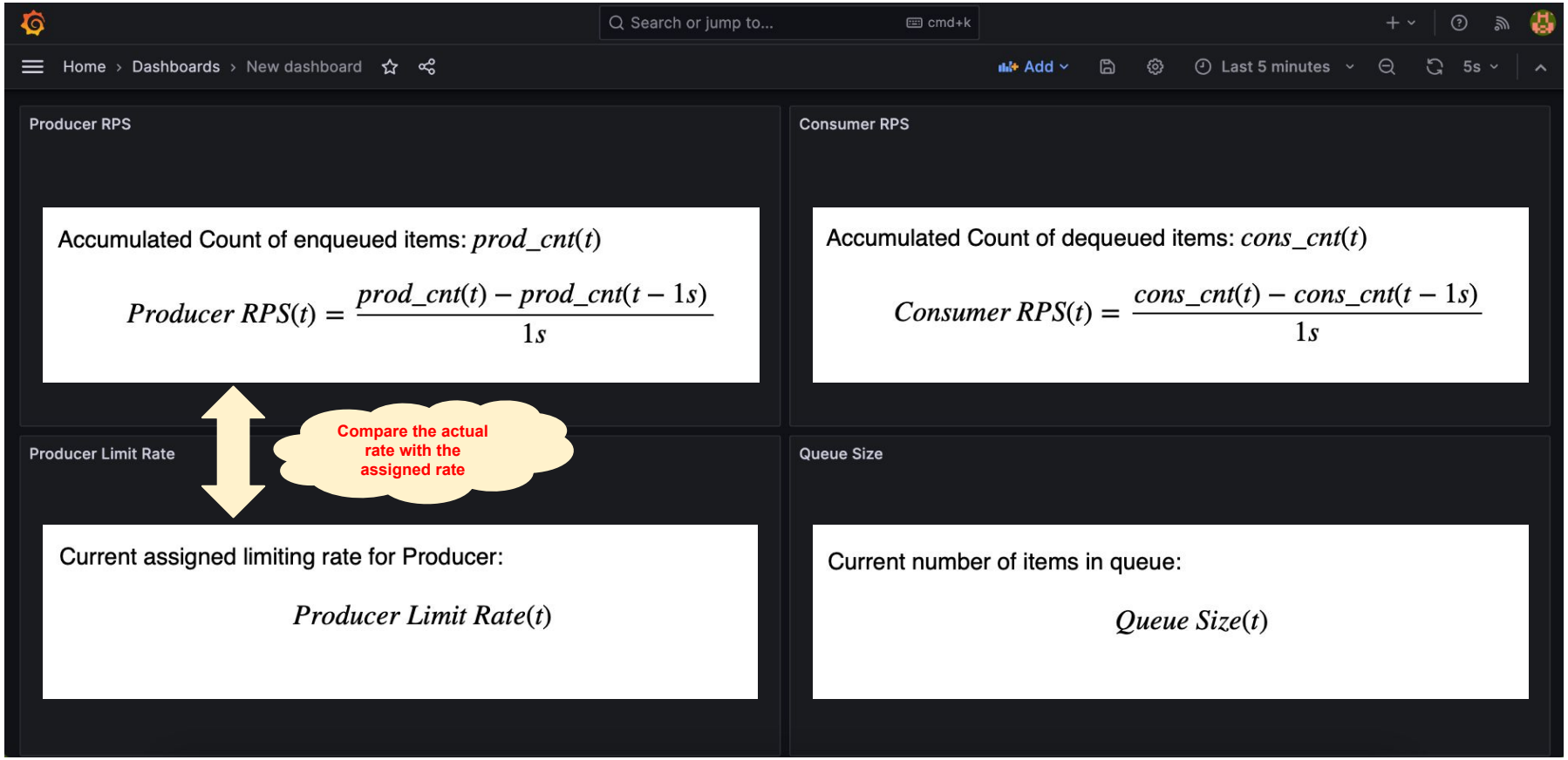
Metrics Monitoring

- **At the instant t**



Metrics Monitoring

- **At the instant t**



04

Results

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Results

Rate Limiter Unit Testing

```
/* Test timing when acquire only one permit */
void test_acquire();

/* Test timing when acquire multiple permits */
void test_acquire_permits();

/* Test tinming when changing to new rate value */
void test_rate_change();
```

Results

Rate Limiter Unit Testing

```c
/* Test timing when acquire only one permit */
void test_acquire();

/* Test timing when acquire multiple permits */
void test_acquire_permits();

/* Test tinming when changing to new rate value */
void test_rate_change();
```

```
(base) nguyencuong6873@MacBookAirNTTC prodcons % ./run.sh
cc test_rate_limiter.c -o test_rate_limiter
elapsed time: 1001
Passed
elapsed time: 4004
Passed
elapsed time: 1000
Passed
elapsed time: 4999
Passed
(base) nguyencuong6873@MacBookAirNTTC prodcons %
```

Results

Rate Limiter Unit Testing

```c
/* Test timing when acquire only one permit */
void test_acquire();

/* Test timing when acquire multiple permits */
void test_acquire_permits();

/* Test tinming when changing to new rate value */
void test_rate_change();
```

```
(base) nguyencuong6873@MacBookAirNTTC prodcons % ./run.sh
cc test_rate_limiter.c -o test_rate_limiter
elapsed time: 1001
Passed
elapsed time: 4004
Passed
elapsed time: 1000
Passed
elapsed time: 4999
Passed
(base) nguyencuong6873@MacBookAirNTTC prodcons %
```

Compare with…

Results

Rate Limiter Unit Testing

```c
/* Test timing when acquire only one permit */
void test_acquire();

/* Test timing when acquire multiple permits */
void test_acquire_permits();

/* Test tinming when changing to new rate value */
void test_rate_change();
```

```
(base) nguyencuong6873@MacBookAirNTTC prodcons % ./run.sh
cc test_rate_limiter.c -o test_rate_limiter
elapsed time: 1001
Passed
elapsed time: 4004
Passed
elapsed time: 1000
Passed
elapsed time: 4999
Passed
(base) nguyencuong6873@MacBookAirNTTC prodcons %
```

Compare with…

google/**guava**

Google core libraries for Java

299 Contributors | 1k Used by | 49k Stars | 11k Forks

guava / guava / src / com / google / common / util / concurrent / **RateLimiter.java**

stefanhaustein and Google Java Core Libraries Internal Build Change 4280533 · last year History

Code Blame 498 lines (461 loc) · 21.5 KB Raw

```
/*
 * Copyright (C) 2012 The Guava Authors
 *
 * Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except
 * in compliance with the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software distributed under the License
 * is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
 * or implied. See the License for the specific language governing permissions and limitations under
 * the License.
 */

package com.google.common.util.concurrent;
```

Results

Rate Limiter Unit Testing

```c
/* Test timing when acquire only one permit */
void test_acquire();

/* Test timing when acquire multiple permits */
void test_acquire_permits();

/* Test tinming when changing to new rate value */
void test_rate_change();
```

```
(base) nguyencuong6873@MacBookAirNTTC prodcons % ./run.sh
cc test_rate_limiter.c -o test_rate_limiter
elapsed time: 1001
Passed
elapsed time: 4004
Passed
elapsed time: 1000
Passed
elapsed time: 4999
Passed
(base) nguyencuong6873@MacBookAirNTTC prodcons %
```

Compare with…

```java
@Test
public void testAcquire();

@Test
public void testAcquirePermits();

@Test
public void testRateChange();
```

Results

Rate Limiter Unit Testing

```c
/* Test timing when acquire only one permit */
void test_acquire();

/* Test timing when acquire multiple permits */
void test_acquire_permits();

/* Test tinming when changing to new rate value */
void test_rate_change();
```

```
(base) nguyencuong6873@MacBookAirNTTC prodcons % ./run.sh
cc test_rate_limiter.c -o test_rate_limiter
elapsed time: 1001
Passed
elapsed time: 4004
Passed
elapsed time: 1000
Passed
elapsed time: 4999
Passed
(base) nguyencuong6873@MacBookAirNTTC prodcons %
```

Compare with…

```java
@Test
public void testAcquire();

@Test
public void testAcquirePermits();

@Test
public void testRateChange();
```

```
(base) nguyencuong6873@MacBookAirNTTC Dev-Demos % ./gradlew src:libs-test:test --tests libs.test.TestGuava

> Task :src:libs-test:test

TestGuava > testAcquire() STANDARD_ERROR
    [Test worker] INFO libs.test.TestGuava - Elapsed Time = 1004

TestGuava > testRateChange() STANDARD_ERROR
    [Test worker] INFO libs.test.TestGuava - Elapsed Time = 1000
    [Test worker] INFO libs.test.TestGuava - Elapsed Time = 4999

TestGuava > testAcquirePermits() STANDARD_ERROR
    [Test worker] INFO libs.test.TestGuava - Elapsed Time = 3999
```

Results

Rate Limiter Unit Testing

```c
/* Test timing when acquire only one permit */
void test_acquire();

/* Test timing when acquire multiple permits */
void test_acquire_permits();

/* Test tinming when changing to new rate value */
void test_rate_change();
```

```
(base) nguyencuong6873@MacBookAirNTTC prodcons % ./run.sh
cc test_rate_limiter.c -o test_rate_limiter
elapsed time: 1001
Passed
elapsed time: 4004
Passed
elapsed time: 1000
Passed
elapsed time: 4999
Passed
(base) nguyencuong6873@MacBookAirNTTC prodcons %
```

Compare with…

google/**guava**

Google core libraries for Java

299 Contributors | 1k Used by | 49k Stars | 11k Forks

- ✔ TOLERANCE < 10 milliseconds with rate = no. permits / second
- ✔ Compare this test results to the the results of testing Rate Limiter from the popular Google Guava Java Library, the TOLERANCE is almost similar

Results

Metrics Visualization

http://localhost:3000/d/f82217e3-90ef-4aa4-a108-492b21bb8839/new-dashboard?orgId=1&refresh=5s&from=now-5m&to=now

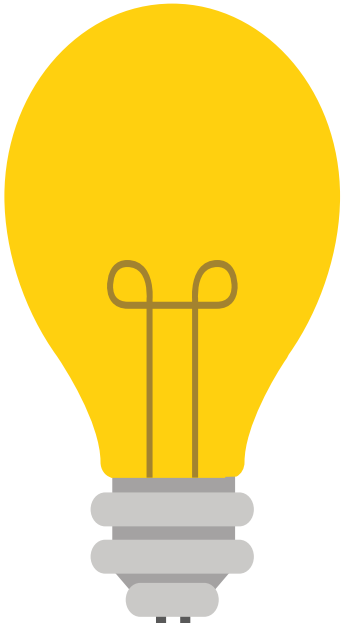
05

Conclusion

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Conclusion



Conclusion

Conclusion 01

Using semaphore and Rate Limiter to control flow in the Producer - Consumer - Orchestrator model

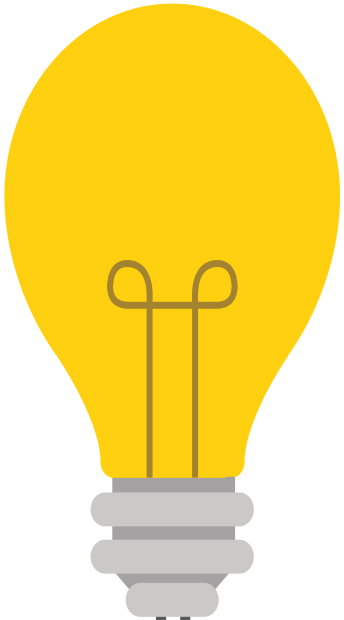
Conclusion

Conclusion 01

Using semaphore and Rate Limiter to control flow in the Producer - Consumer - Orchestrator model

Conclusion 02

Using Token Bucket Algorithm for Rate Limiter implementation



Conclusion

Conclusion 01

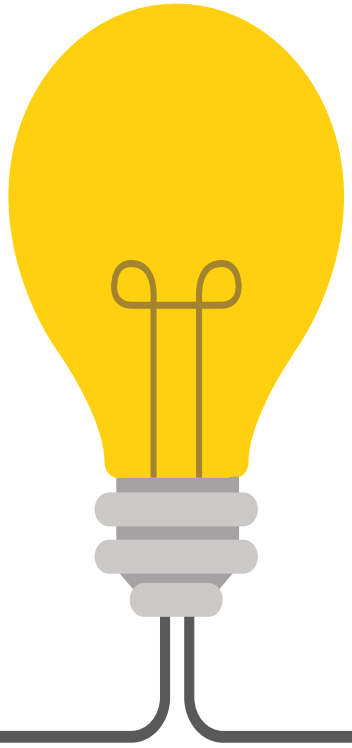
Using semaphore and Rate Limiter to control flow in the Producer - Consumer - Orchestrator model

Conclusion 02

Using Token Bucket Algorithm for Rate Limiter implementation

Conclusion 03

Using Prometheus and Grafana for the purpose of visualization with the support of Java Client Library



UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Thank You