Федеральное агентство связи

Федеральное государственное бюджетное образовательное

учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра ВС

Курсовая работа
по дисциплине «Моделирование»
на тему
«Реализация монитора событий»

Выполнил:
студент гр. МГ-165
Терешков Р. В.

Проверил:
д.т.н., доцент
Родионов А. С.

Новосибирск – 2017

Задание

В рамках данного курсового проекта необходимо реализовать монитор событий для модели, заданной следующим образом: в системе имеется два устройства, первое A имеет интенсивность обслуживания λ, значения для которой задаются с помощью экспоненциального распределения, второе устройство B имеет интенсивность обслуживания λ/k (т.е. работает в k раз медленнее первого). На вход устройствам равномерно поступает поток требований τ. Устройство A имеет ограниченную очередь задач, состоящую из N элементов. Очередь на устройстве B является бесконечной. Более приоритетным является первое из устройств. Если очередь на устройстве A переполнена, задачи начинают поступать на устройство B. Если же, наоборот, пуста, и все задачи были распределены, то на устройство A начинают поступать задачи с устройства B.

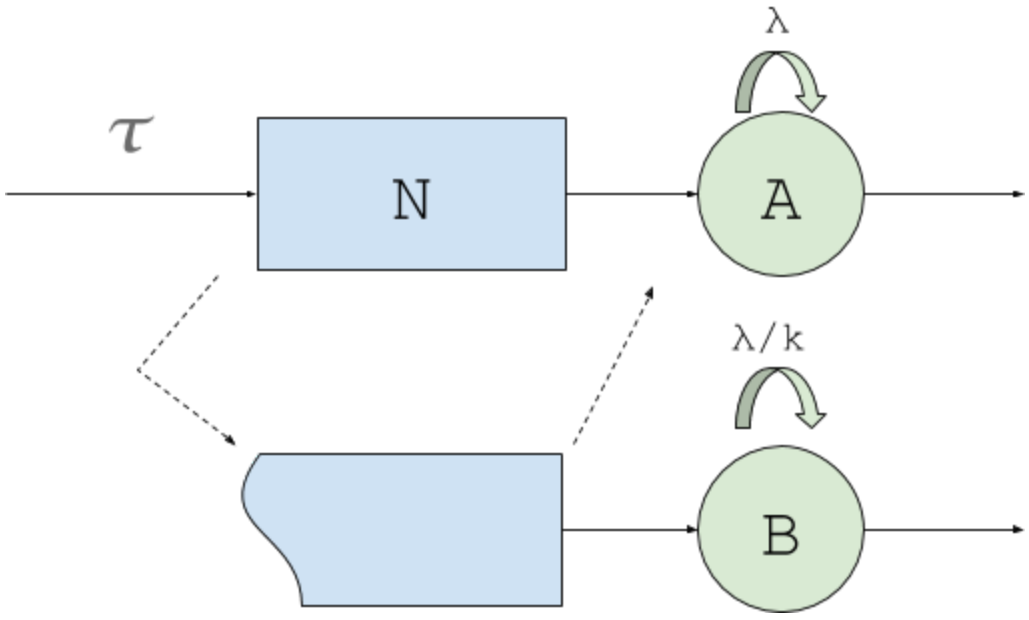


Рис. 1 — Модель для проектирования

Входные данные: λ = 1.0 (exp_distr); τ = 0.7...1.1 (uni_distr); N = 6.

Результатом работы также является исследование среднего времени пребывания требования в системе и подсчёт коэффициентов использования приборов в зависимости от k = 2…10 при заданных значениях распределениях потока, обслуживания и размера очереди N для первого устройства.

Ход работы

В ходе выполнения курсовой работы была реализована программа на языке программирования C++, которая осуществляет мониторинг смоделированных событий. Каждое из событий помещается в двусвязный список — календарь событий. Структура календаря, состоящего из трёх событий, проиллюстрирована ниже:

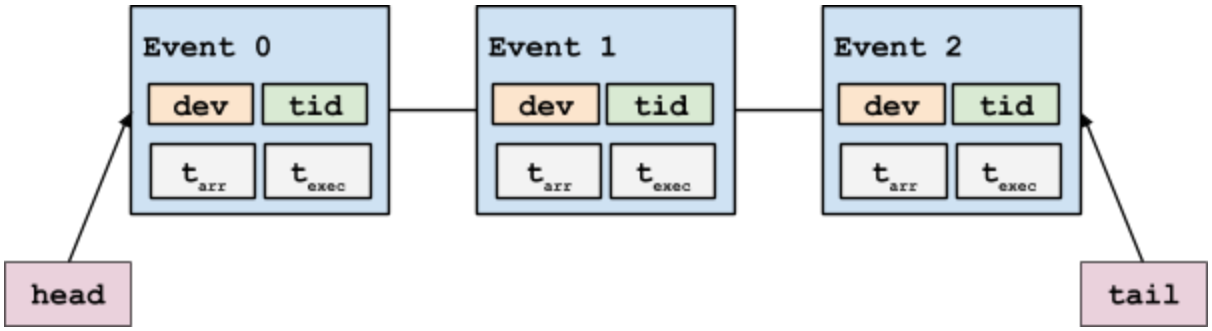


Рис. 2 — Пример календаря, состоящего из трёх событий

Причём события в календаре упорядочены в порядке возрастания по параметру t_{exec}, обозначающего время выполнения этого события. Поле t_{arr} обозначает время прибытия требования в систему. Поле `dev` отвечает тип устройства, на котором будет обработано требование, принадлежащее данному событию: быстрое (`dev = fast`) или медленное (`dev = slow`). `tid` является идентификатором требования.

Генерирование величины `lm` происходит по экспоненциальному закону распределения вероятности с заданным параметром λ:

$$lm = -(\ln(1 - \xi) / \lambda)$$

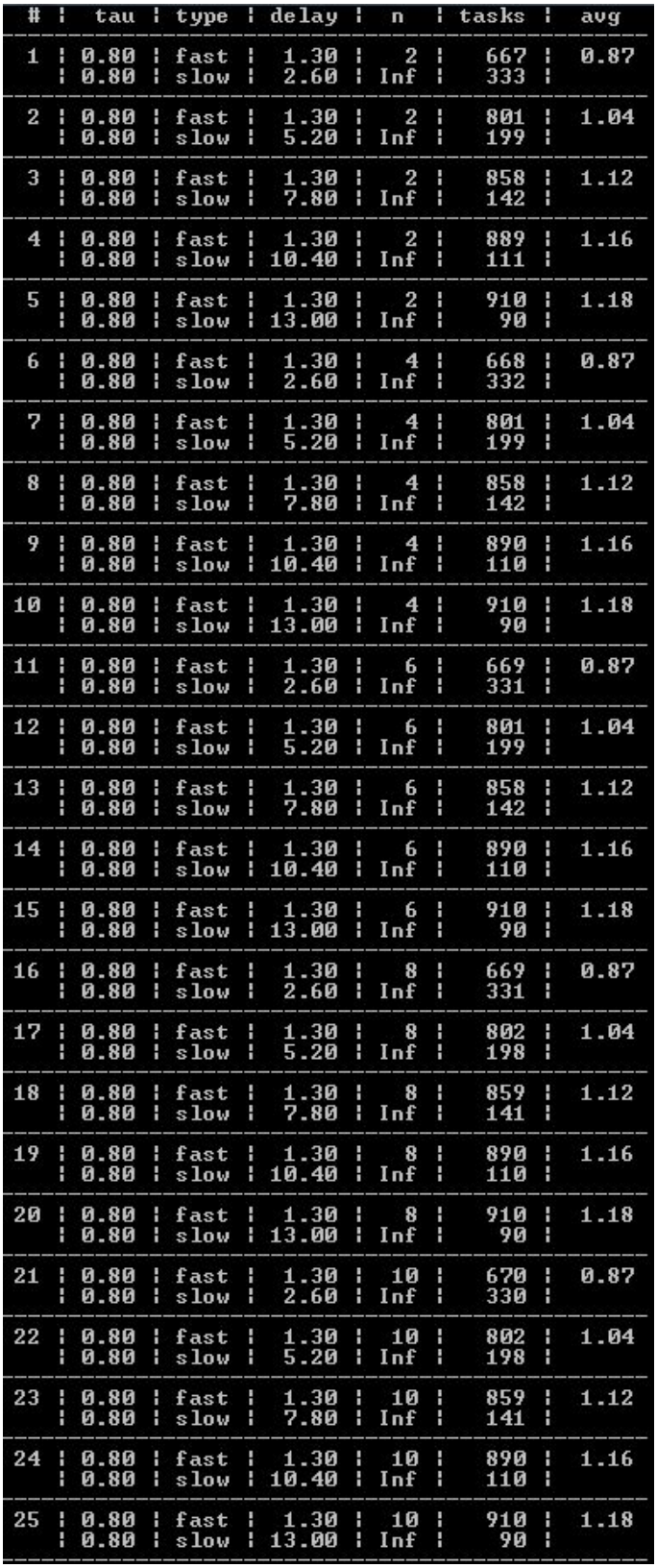
Генерирование величины `dt` происходит по равномерному закону распределения вероятности с диапазоном значений от 0.7 до 1.1:

$$dt = \xi * (1.1 - 0.7) + 0.7$$

Результаты моделирования

#	tau	type	delay	n	tasks	avg
1	0.76	fast	0.18	6	1000	0.76
	0.76	slow	0.36	Inf	0	
2	0.76	fast	0.50	6	1000	0.76
	0.76	slow	1.00	Inf	0	
3	0.76	fast	1.29	6	669	0.86
	0.76	slow	2.58	Inf	331	
4	0.76	fast	1.56	6	669	1.04
	0.76	slow	3.12	Inf	331	
5	0.76	fast	2.94	6	669	1.97
	0.76	slow	5.88	Inf	331	
6	0.97	fast	0.18	6	1000	0.97
	0.97	slow	0.36	Inf	0	
7	0.97	fast	0.50	6	1000	0.97
	0.97	slow	1.00	Inf	0	
8	0.97	fast	1.29	6	756	0.97
	0.97	slow	2.58	Inf	244	
9	0.97	fast	1.56	6	669	1.04
	0.97	slow	3.12	Inf	331	
10	0.97	fast	2.94	6	669	1.97
	0.97	slow	5.88	Inf	331	
11	1.02	fast	0.18	6	1000	1.02
	1.02	slow	0.36	Inf	0	
12	1.02	fast	0.50	6	1000	1.02
	1.02	slow	1.00	Inf	0	
13	1.02	fast	1.29	6	799	1.03
	1.02	slow	2.58	Inf	201	
14	1.02	fast	1.56	6	669	1.04
	1.02	slow	3.12	Inf	331	
15	1.02	fast	2.94	6	669	1.97
	1.02	slow	5.88	Inf	331	
16	1.03	fast	0.18	6	1000	1.03
	1.03	slow	0.36	Inf	0	
17	1.03	fast	0.50	6	1000	1.03
	1.03	slow	1.00	Inf	0	
18	1.03	fast	1.29	6	805	1.04
	1.03	slow	2.58	Inf	195	
19	1.03	fast	1.56	6	669	1.04
	1.03	slow	3.12	Inf	331	
20	1.03	fast	2.94	6	669	1.97
	1.03	slow	5.88	Inf	331	
21	1.10	fast	0.18	6	1000	1.10
	1.10	slow	0.36	Inf	0	
22	1.10	fast	0.50	6	1000	1.10
	1.10	slow	1.00	Inf	0	
23	1.10	fast	1.29	6	856	1.10
	1.10	slow	2.58	Inf	144	
24	1.10	fast	1.56	6	708	1.11
	1.10	slow	3.12	Inf	292	
25	1.10	fast	2.94	6	669	1.97
	1.10	slow	5.88	Inf	331	

Рис. 3 — Поведение системы в зависимости от различных значений lm и dt



#	tau	type	delay	n	tasks	avg
1	0.80	fast	1.30	2	667	0.87
	0.80	slow	2.60	Inf	333	
2	0.80	fast	1.30	2	801	1.04
	0.80	slow	5.20	Inf	199	
3	0.80	fast	1.30	2	858	1.12
	0.80	slow	7.80	Inf	142	
4	0.80	fast	1.30	2	889	1.16
	0.80	slow	10.40	Inf	111	
5	0.80	fast	1.30	2	910	1.18
	0.80	slow	13.00	Inf	90	
6	0.80	fast	1.30	4	668	0.87
	0.80	slow	2.60	Inf	332	
7	0.80	fast	1.30	4	801	1.04
	0.80	slow	5.20	Inf	199	
8	0.80	fast	1.30	4	858	1.12
	0.80	slow	7.80	Inf	142	
9	0.80	fast	1.30	4	890	1.16
	0.80	slow	10.40	Inf	110	
10	0.80	fast	1.30	4	910	1.18
	0.80	slow	13.00	Inf	90	
11	0.80	fast	1.30	6	669	0.87
	0.80	slow	2.60	Inf	331	
12	0.80	fast	1.30	6	801	1.04
	0.80	slow	5.20	Inf	199	
13	0.80	fast	1.30	6	858	1.12
	0.80	slow	7.80	Inf	142	
14	0.80	fast	1.30	6	890	1.16
	0.80	slow	10.40	Inf	110	
15	0.80	fast	1.30	6	910	1.18
	0.80	slow	13.00	Inf	90	
16	0.80	fast	1.30	8	669	0.87
	0.80	slow	2.60	Inf	331	
17	0.80	fast	1.30	8	802	1.04
	0.80	slow	5.20	Inf	198	
18	0.80	fast	1.30	8	859	1.12
	0.80	slow	7.80	Inf	141	
19	0.80	fast	1.30	8	890	1.16
	0.80	slow	10.40	Inf	110	
20	0.80	fast	1.30	8	910	1.18
	0.80	slow	13.00	Inf	90	
21	0.80	fast	1.30	10	670	0.87
	0.80	slow	2.60	Inf	330	
22	0.80	fast	1.30	10	802	1.04
	0.80	slow	5.20	Inf	198	
23	0.80	fast	1.30	10	859	1.12
	0.80	slow	7.80	Inf	141	
24	0.80	fast	1.30	10	890	1.16
	0.80	slow	10.40	Inf	110	
25	0.80	fast	1.30	10	910	1.18
	0.80	slow	13.00	Inf	90	

Рис. 4 — Поведение системы в зависимости от различных значений N и k

dev	task	arrived	executed
fast	1	0.50	0.50
fast	2	1.00	1.50
fast	3	1.50	2.50
fast	4	2.00	3.50
fast	5	2.50	4.50
slow	11	5.50	5.50
fast	6	3.00	5.50
fast	7	3.50	6.50
slow	13	6.50	7.50
fast	8	4.00	7.50
fast	9	4.50	8.50
slow	15	7.50	9.50
fast	10	5.00	9.50
fast	12	6.00	10.50
slow	17	8.50	11.50
fast	14	7.00	11.50
fast	16	8.00	12.50
slow	19	9.50	13.50
fast	18	9.00	13.50
fast	20	10.00	14.50
slow	21	10.50	15.50
fast	22	11.00	15.50
fast	24	12.00	16.50
slow	23	11.50	17.50
fast	26	13.00	17.50
fast	28	14.00	18.50
slow	25	12.50	19.50
fast	30	15.00	19.50
fast	32	16.00	20.50
slow	27	13.50	21.50
fast	34	17.00	21.50
fast	36	18.00	22.50
slow	29	14.50	23.50
fast	38	19.00	23.50
fast	40	20.00	24.50
slow	31	15.50	25.50
fast	42	21.00	25.50
fast	44	22.00	26.50
slow	33	16.50	27.50
fast	46	23.00	27.50
fast	48	24.00	28.50
slow	35	17.50	29.50
fast	50	25.00	29.50
fast	52	26.00	30.50
slow	37	18.50	31.50
fast	54	27.00	31.50
fast	56	28.00	32.50
slow	39	19.50	33.50
fast	58	29.00	33.50
fast	60	30.00	34.50
slow	41	20.50	35.50

Рис. 5 — Календарь событий при dt = 0.5, lm = 1.0, N = 6, k = 2

Вывод

При увеличении значения потока требований `dt`, либо при увеличении задержки на обслуживание требования `lm` и при фиксированных `N` и `k`, среднее время нахождения требования в системе возрастает.

При фиксированных значениях `dt` и `lm` и при увеличении значения `N` скорость работы системы не изменяется. При увеличении параметра `k` устройства `B`, большую нагрузку берёт на себя устройство `A`.

Листинг программы

device.h

```
#ifndef _DEVICE_H_
#define _DEVICE_H_

#include <memory>
#include <string>
#include <queue>

struct Task {
	Task(int _id, double _t)
		: id(_id)
		, arrived_time(_t) {}

	int id;
	double arrived_time;
};

class Device {
public:
	Device();
	Device(std::string, double);

	std::string GetType() const { return type; }
	double GetDelay() const { return delay; }
	double GetLastCompletedTaskTime() const { return lct_time; }
	int GetCompletedTasksCount() const { return ct_counter; }
	const std::queue<Task> & GetTaskQueue() const { return taskQueue; }

	void IncreaseCompletedTasksCounter() { ++ct_counter; }
	void IncreaseProgressTime(double _t) { progress_time += _t; }

	bool Progress();

	void Dequeue() { taskQueue.pop(); }
	void Enqueue(Task _task) { taskQueue.push(_task); }

private:
	std::string type;
	double delay;
	double progress_time;
	double lct_time;
	int ct_counter;

	std::queue<Task> taskQueue;
};

#endif
```

device.cpp

```
#include <iostream>
#include <queue>

#include "device.h"

Device::Device()
	: type("slow")
	, delay(0.0)
	, progress_time(0)
	, lct_time(0)
	, ct_counter(0) {}

Device::Device(std::string _t, double _delay)
	: type(_t)
	, delay(_delay)
	, progress_time(0)
	, lct_time(0)
	, ct_counter(0) {}

bool Device::Progress()
{
	if (!taskQueue.empty())
	{
		if (progress_time > delay)
		{
			progress_time -= delay;
			lct_time = (lct_time) ? lct_time + delay :
taskQueue.front().arrived_time;
			return true;
		}
	}

	return false;
}
```

schedule.h

```
#ifndef _SCHEDULE_H_
#define _SCHEDULE_H_

#include <list>
#include <memory>

struct Event {
	Event(std::string _dtype, int _tid, double _at, double _et)
		: device_type(_dtype)
		, task_id(_tid)
		, arrive_time(_at)
		, execution_time(_et) {}

	std::string device_type;
```

```
        int task_id;
        double arrive_time;
        double execution_time;
};

class Schedule
{
public:
        Schedule()
        {
                schedule = std::list<Event>();
        }

        const std::list<Event> & Get() { return schedule; }

        void AddEvent(Event _e) { schedule.push_back(_e); }
        void RemEvent() { schedule.pop_back(); }

        void Sort();
        void PrintSchedule();

private:
        std::list<Event> schedule;
};

#endif
```

---

schedule.cpp

```
#include <algorithm>
#include <iostream>
#include <iomanip>
#include <string>
#include <queue>

#include "schedule.h"

void Schedule::Sort()
{
        schedule.sort([](const Event & _e1, const Event & _e2)
        {
                return _e1.execution_time < _e2.execution_time;
        });
}

void Schedule::PrintSchedule()
{
        std::cout << "\n   dev | task | arrived | executed \n";
        std::cout << "-------------------------------------\n";

        for (const auto & e : schedule)
        {
                std::cout << std::fixed << "  " << e.device_type << " | "
                        << std::setw(4) << e.task_id << " | "
```

```
                        << std::setw(7) << std::setprecision(2) << e.arrive_time << " | "
                        << std::setw(7) << std::setprecision(2) << e.execution_time << "\n";
        }
}
```

---

main.cpp

```
#include <algorithm>
#include <cstdlib>
#include <iomanip>
#include <iostream>
#include <cmath>
#include <string>
#include <thread>

#include "device.h"
#include "schedule.h"

#define TASKS_COUNT 100

struct Stats {
        Stats(std::string _dtype, double _uptime, double _dt, double _delay, int _n, int _ct,
int _id)
                : dtype(_dtype), uptime(_uptime), dt(_dt), delay(_delay), n(_n), ct(_ct),
id(_id) {}

        std::string dtype;

        double uptime;
        double dt;
        double delay;

        int n;
        int ct;

        int id;
};

double exp_distr(double, double);
double uni_distr(double, double);

void DoProgress(std::vector<std::shared_ptr<Device>> & devices, double dt,
std::shared_ptr<Schedule> schedule);
void PrintStatistics(std::vector<std::shared_ptr<std::pair<Stats, Stats>>> & statistics);

int main()
{
        srand((unsigned)time(NULL));

        int iter = 1;
        int k = 2;
        size_t n = 6;

        auto devices = std::vector<std::shared_ptr<Device>>();
```

```
auto statistics = std::vector<std::shared_ptr<std::pair<Stats, Stats>>>();

auto tau = std::vector<double>();

for (int i = 0; i < iter; ++i)
{
	auto dt = uni_distr(0.7, 1.1);
	tau.push_back(dt);
}

std::sort(tau.begin(), tau.end());

auto lambda = std::vector<double>();

for (int i = 0; i < iter; ++i)
{
	auto ksi = uni_distr(0.0, 1.0);
	auto dt = exp_distr(ksi, 0.8);
	lambda.push_back(dt);
}

std::sort(lambda.begin(), lambda.end());

int exp = 0;

for (auto dt : tau)
{
	for (auto lm : lambda)
	{
		double uptime = 0.0;

		devices.push_back(std::make_shared<Device>("fast", lm));
		devices.push_back(std::make_shared<Device>("slow", lm * k));

		auto fast = devices[0];
		auto slow = devices[1];

		auto schedule = std::make_shared<Schedule>();

		// Spread tasks
		for (int i = 1; i <= TASKS_COUNT; ++i)
		{
			if (fast->GetTaskQueue().size() < n)
			{
				fast->Enqueue(Task(i, dt * i));
			}
			else
			{
				slow->Enqueue(Task(i, dt * i));
			}

			DoProgress(devices, dt, schedule);

			uptime += dt;
		}

		// Wait 'til devices complete their work
```

```
                        while (schedule->Get().size() != TASKS_COUNT)
                        {
                                if (fast->GetTaskQueue().size() < n)
                                {
                                        while (!slow->GetTaskQueue().empty() &&
fast->GetTaskQueue().size() < n)
                                        {
                                                fast->Enqueue(slow->GetTaskQueue().front());
                                                slow->Dequeue();
                                        }
                                }

                                DoProgress(devices, dt, schedule);

                                uptime += dt;
                        }

                        ++exp;

                        auto fast_stat = Stats(fast->GetType(), uptime, dt, fast->GetDelay(), n,
fast->GetCompletedTasksCount(), exp);
                        auto slow_stat = Stats(slow->GetType(), uptime, dt, slow->GetDelay(),
-1, slow->GetCompletedTasksCount(), exp);
                        auto stat = std::make_shared<std::pair<Stats,
Stats>>(std::make_pair(fast_stat, slow_stat));
                        statistics.push_back(stat);

                        schedule->Sort();
                        schedule->PrintSchedule();

                        devices.clear();
                }
        }

        PrintStatistics(statistics);

        getchar();
        return 0;
}

double exp_distr(double ksi, double lambda)
{
        auto number = -1.0 * (log(1.0 - ksi) / lambda);
        return number;
}

double uni_distr(double a, double b)
{
        auto number = ((double)rand() / (RAND_MAX)) * (b - a) + a;
        return number;
}

void DoProgress(std::vector<std::shared_ptr<Device>> & devices, double dt,
std::shared_ptr<Schedule> schedule)
{
        for (auto & device : devices)
        {
```

```
                device->IncreaseProgressTime(dt);

                if (device->Progress())
                {
                        schedule->AddEvent(Event(device->GetType(),
device->GetTaskQueue().front().id, device->GetTaskQueue().front().arrived_time,
device->GetLastCompletedTaskTime()));
                        device->Dequeue();
                        device->IncreaseCompletedTasksCounter();
                }
        }
}

void PrintStatistics(std::vector<std::shared_ptr<std::pair<Stats, Stats>>> & statistics)
{
        std::cout << "  # |  tau | type | delay |  n  | tasks |  avg  \n";
        std::cout << "-----------------------------------------------\n";

        for (const auto & stat : statistics)
        {
                std::cout << std::fixed << std::setw(3) << stat->first.id << " | "
                        << std::setw(4) << std::setprecision(2) << stat->first.dt << " | "
                        << std::setw(4) << stat->first.dtype << " | "
                        << std::setw(5) << std::setprecision(2) << stat->first.delay << " | "
                        << std::setw(3) << stat->first.n << " | "
                        << std::setw(5) << stat->first.ct << " | "
                        << std::setw(5) << std::setprecision(2) << stat->second.uptime /
TASKS_COUNT << "\n";

                std::cout << std::fixed << std::setw(3) << " " << " | "
                        << std::setw(4) << std::setprecision(2) << stat->second.dt << " | "
                        << std::setw(4) << stat->second.dtype << " | "
                        << std::setw(5) << std::setprecision(2) << stat->second.delay << " | "
                        << std::setw(3) << "Inf" << " | "
                        << std::setw(5) << stat->second.ct << " | "
                        << std::setw(5) << std::setprecision(2) << "\n";

                std::cout << "-----------------------------------------------\n";
        }
}
```
