
Universidade Federal do Rio de
Janeiro
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

**TRABALHO FINAL
AVALIAÇÃO EM DESEMPENHO**

Professor Paulo Aguiar

Sumario

0.1	Introdução	pag 3
0.1.1	Funcionamento geral do simulador	pag 3
0.1.2	Linguagem utilizada e escolha	pag 3
0.1.3	Estruturas internas e eventos	pag 4
0.1.4	Forma de geração de variáveis aleatórias	pag 5
0.1.5	Método de simulação	pag 5
0.1.6	Final da rodada	pag 5
0.1.7	Escolha dos parâmetros	pag 5
0.1.8	Máquinas usadas	pag 6
0.1.9	Outras informações	pag 6
0.2	Teste de Correção	pag 6
0.2.1	sample mean test	pag 6
0.2.2	data package test	pag 7
0.2.3	voice package number test	pag 8
0.2.4	event list test	pag 8
0.2.5	queue test	pag 9
0.2.6	round test	pag 10
0.3	Estimativa de Fase Transiente	pag 21
0.3.1	Para $\rho=0.1$ (não-preemptivo)	pag 21
0.3.2	Para $\rho=0.2$ (não-preemptivo)	pag 24
0.3.3	Para $\rho=0.3$ (não-preemptivo)	pag 26
0.3.4	Para $\rho=0.4$ (não-preemptivo)	pag 29
0.3.5	Para $\rho=0.5$ (não-preemptivo)	pag 31
0.3.6	Para $\rho=0.6$ (não-preemptivo)	pag 34
0.3.7	Para $\rho=0.7$ (não-preemptivo)	pag 36
0.3.8	Para $\rho=0.1$ (preemptivo)	pag 39
0.3.9	Para $\rho=0.2$ (preemptivo)	pag 41
0.3.10	Para $\rho=0.3$ (preemptivo)	pag 43
0.3.11	Para $\rho=0.4$ (preemptivo)	pag 46
0.3.12	Para $\rho=0.5$ (preemptivo)	pag 48
0.3.13	Para $\rho=0.6$ (preemptivo)	pag 50
0.3.14	Para $\rho=0.7$ (preemptivo)	pag 53

0.4	Listagem Documentada do Programa	pag 55
0.4.1	simulator.cpp	pag 55
0.4.2	run queues.cpp	pag 56
0.4.3	Customer.cpp	pag 67
0.4.4	Customer.h	pag 68
0.4.5	data structures.cpp	pag 69
0.4.6	data structures.h	pag 71
0.4.7	Event.cpp	pag 72
0.4.8	Event.h	pag 74
0.4.9	statistics.cpp	pag 75
0.4.10	statistics.h	pag 77
0.4.11	test correction.cpp	pag 77
0.5	Tabela com Resultados	pag 92
0.5.1	Casos Sem Preempção	pag 92
0.5.2	Casos Com Preempção	pag 94
0.5.3	Sementes diferentes	pag 97
0.6	Conclusões	pag 98

Trabalho Final de Avaliação em Desempenho - MAB 515

Nomes: Ingrid Quintanilha Pacheco e Júlio Rama Krsna Mandoju

Julho, 2018

1 Introdução

O trabalho passado para conclusão do curso foi o desenvolvimento de um simulador (roteador) com duas filas e um servidor. A primeira fila se reserva para a chegada de pacotes de dados e a segunda para pacotes de voz dos 30 canais independentes, que possuem prioridade sobre o primeiro tipo mencionado. A chegada de um pacote de voz pode implicar, ou não, na interrupção de um pacote de dados que esteja sendo servido, com o serviço preemptivo ou não. Além disso, a disciplina de atendimento é FCFS (FIFO), fazendo com que pacotes de voz de canais diferentes possam se intercalar, devido a ordem de chegada e o término do período de silêncio que cada canal possui, só transmitindo pacotes no seu período de atividade.

1.1 Funcionamento geral do simulador

O simulador funciona como explicado na descrição do trabalho e na especificação da apostila. De uma forma geral, recebe uma fila única de eventos, no qual toda chegada (pode ser pacote de dados ou voz) ao sistema, saída do servidor e fim do período de silêncio de um canal de voz é um evento e será inserido em ordem cronológica. O evento da fila a ser tratado é sempre o primeiro, causando a adição de novos eventos à fila. Trabalha com prioridade do pacote de voz sobre dados e com e sem preempção com que diz relação ao atendimento no servidor.

1.2 Linguagem utilizada e escolha

A linguagem utilizada no trabalho foi C++, pelo fato de ser a linguagem que ambos os membros da equipe tinham mais domínio, fazendo com que o trabalho pudesse ser melhor aproveitado, além de ser uma linguagem que permite orientação a objeto, facilitando a criação de classes, trazendo versatilidade para o seu uso. Outro motivo para a escolha foi o desempenho que ela proporciona e a portabilidade, apesar da complexidade para gerar gráficos, porque não possui uma biblioteca específica para isso. Nesse caso, precisamos gerar um arquivo que contivesse os dados necessários para gerações dos gráficos e implementamos

um programa simples em Python para a plotagem dos gráficos com a biblioteca Matplotlib.

1.3 Estruturas internas, eventos utilizados e explicacao detalhada de como foi implementada a lista de eventos

Como explicado acima, no programa temos 5 tipos de eventos:

- Chegada de pacote de voz
- Chegada de pacote de dados
- Saída de pacote de voz
- Saída de pacote de dados
- Fim de período de silêncio

Cada um desses eventos pertence à classe Evento, e a lista de eventos é uma lista encadeada no qual cada item pertencente é da classe Evento. Quando o programa começa, temos a criação de um evento de chegada de dados e do fim de período de silêncio de cada canal, sendo logo após inseridos na lista. A inserção na lista é feita com ordenação cronológica, logo, o evento que ocorrer antes (o pacote de dados ou o fim do período de silêncio) é posto mais a frente na lista, e será o primeiro a ser tratado. Já no tratamento de cada evento, cada evento tem um método `treat_event` que atualiza as filas de espera e o servidor de acordo. Além disso, a chegada de um pacote de voz insere outro evento de chegada de voz na lista (como descrito na apostila) OU o final do próximo período de silêncio do seu respectivo canal, caso seja o último pacote do atual período ativo. Uma chegada de um pacote de dados insere outro evento de chegada de dados na lista. Se o evento for o fim de um período de silêncio, é gerado um número com distribuição geométrica para o número de pacotes que o novo período ativo produzirá, e depois ele insere a chegada do primeiro pacote do período de atividade na lista, com o timestamp sendo o mesmo do evento de fim (pacote chega na hora que o período de silêncio termina). Se o evento for a saída de um pacote do sistema, então só são calculadas as suas métricas. Todo evento, depois de tratado, é tirado da lista e passa a apontar para o próximo.

Além dos eventos, temos também uma classe de fregueses chamada Customer. Cada pacote é um customer, e eles têm que ser gerados antes dos eventos que os recebem como parâmetro. Os customers são usados no tratamento dos eventos, para identificação do fregues que está sendo servido no momento e decisão (dependendo de ter ou não preempção) do usuário que vai ser servido (interrupção de um pacote de dados caso tenha a chegada de um pacote de voz e o serviço seja preemptivo).

Além dessa classes, temos estruturas que servem para auxílio do servidor, como as filas QUEUE de pacotes de dados e voz, e o array de canais de voz, que guarda quantos pacotes de voz restam a serem criados por cada canal até que ele entre em período de silêncio.

```
while ((round > 0 && Customer::total_customers < customers_number * (round)) ||
       (round == 0 && Customer::total_customers < 0)) {
```

Os intervalos, que são usados para cálculo das métricas, são armazenados em um vetor, e o `last_time`, que é o último tempo que algum pacote do determinado canal entrou no servidor, serve para calcular o intervalo entre os pacotes do mesmo período ativo do mesmo canal.

1.4 A forma de geracao das variaveis aleatorias envolvidas

Na situação apresentada, foi necessária a criação de números aleatórios, tanto para o tempo de chegada de um pacote de dados, quanto para o tamanho do mesmo, ou até o período de silêncio de um canal de voz. A biblioteca "stdlib" nos fornece a função "rand()", que gera randomicamente um número, e as variáveis aleatórias geradas no programa se alimentam dele como parâmetro para cálculo dos seus valores. Além disso, para trazer mais dinamicidade para essa criação, recebemos um valor inicial de semente como input e setamos ele no gerador através da função "srand()", fazendo com que o próprio usuário possa escolher com que valor iniciaremos a geração dos números aleatórios.

1.5 Método de simulação

O método escolhido foi o batch, pois ele apresenta a facilidade de as rodadas não se atravessarem com outro período transiente. Além disso, a possibilidade de fregueses de uma rodada impactar na outras (no tempo de intervalo entre pacotes do mesmo período ativo cortados por rodadas diferentes) não trouxe problema (pois estão sendo calculados de forma independente), então achamos mais vantajoso esse método. A semente é descrita acima, recebida por input e setada no começo do programa. Como as métricas de cada rodada são separadas, elas não influenciam nas métricas umas das outras garantindo independencia nas rodadas.

1.6 Final da rodada

A quantidade de fregueses por rodada é recebida como input no começo do programa. Através dessa quantidade há uma verificação:

Essa verificação se dá de forma que se a rodada não for o período transiente, ela ocorre até que a quantidade de fregueses por rodada tenha sido atingida, já se for o período transiente, roda até a quantidade estipulada de amostras na fase trasiente acabar e a as rodadas reais começarem.

1.7 Escolha dos parâmetros

Os parâmetros, como número de fregueses por rodada e quantidade de rodadas, são recebidos como entrada pelo usuário e utilizados para a execução das rodadas

e extrações das métricas necessárias. A escolha do período transiente ideal para cada caso será vista mais a fundo na seção 3. Quanto maior o número de rodadas, menor tende a ser o desvio padrão das médias entre rodadas, visto que haverá mais médias a serem comparadas, porém um número muito baixo de fregueses por rodada costuma ser ruim, visto que diminui o total de amostras utilizadas.

1.8 Máquinas usadas

O programa foi rodado em duas máquinas:

- Um Macbook Pro (13-inch, Mid 2009) - Processador 2,53 GHz Intel Core 2 Duo - Memória 8GB 1333 MHz DDR3 - Gráficos NVIDIA GeForce 9400M 256MB - Sistema Operacional: OSX El Capitan (Versão 10.11.6)
- Um Desktop com processador Intel(R) Core(TM) i7-6700 CPU @ 3.40 GHz 3.41 GHz - Memória 8 GB - Sistema operacional 64 bits, com processador com base em x64 - Windows 10 Home Single Language

1.9 Outras Informações Pertinentes

O código fonte do trabalho pode ser encontrado no seguinte endereço:

- <https://github.com/ingridpacheco/trabalho-de-ad>

2 Teste de Correção

Para ter certeza de que o programa está funcionando corretamente, foi-se criado um arquivo de testes: `test_correction.cpp`. Ele possui diversos testes, que são:

2.1 sample mean test

```
void sample_mean_test(int (*function)()) {
    int current_sample;
    int n64 = 0;
    int n512 = 0;
    int n1500 = 0;
    for (int i = 0; i < BIGNUM; i++) {
        current_sample = (*function)();
        if (current_sample == 64) n64++;
        else if (current_sample == 512) n512++;
        else if (current_sample == 1500) n1500++;
    }
    cout << "\t\t64: " << (double)n64/BIGNUM * 100 << "%\n\t\t512: " <<
    (double)n512/BIGNUM * 100 << "%\n\t\t1500: " << (double)n1500/BIGNUM * 100 << "%" << endl;
}
```

Nesse teste, verificamos a porcentagem na qual pacotes de dados de tamanho 64, 512 e 1500 aparecem. De acordo com o pdf, as porcentagens deveriam ser:
 $D(u) =$

$$\begin{aligned} \{ & 64 : \text{if } 0 \leq u < 0.3 \\ & 512 : \text{if } 0.3 \leq u < 0.4 \\ & 1500 : \text{if } 0.4 \leq u < 0.7 \\ & \frac{14360}{3}u - 9860 \text{ if } u \geq 0.7 \end{aligned}$$

De acordo, com o resultado do teste:

Sample Mean: 755.129 bytes

- 64: 30.0391%
- 512: 10.018%
- 1500: 29.9702%

Logo, as porcentagens estão extremamente próximas do esperado, confirmando que os tamanhos estão como esperados. Além disso, a média também está muito perto do valor colocado no PDF 755 bytes, também apresentando corretude.

2.2 data package size test

```
// Function that determines the size of a data package (actual function)
int data_package_size_test() {
    long double seed = rand1();
    if (seed < .3) return 64;
    else if (seed < .3 + .1) return 512;
    else if (seed < .3 + .1 + .3) return 1500;
    else {
        seed = (seed - .3 - .1 - .3) / (1 - .3 - .1 - .3);
        return seed * (1500 - 64) + 64;
    }
}
```

Esse é o teste utilizado dentro do teste acima para verificar as porcentagens. Nele, através de uma semente gerada randomicamente temos o tamanho do pacote de dados criado, e a sua aplicação no teste acima comprova que está correto.

2.3 voice package number test

```
// Function that determines the number of voice packages (actual function)
int voice_package_number_test() {
    long double seed = (long double) rand();
    if (seed == 1) seed = .999999; // seed cannot be 1 or there would be a log(0), which is infinite,
    // so... we make it a little smaller
    return (int) ceil(log(1.0 - seed) / log(1 - 1.0L / 22.0));
}
```

Nesse teste é testada a média do número de pacotes de voz que é criado para um canal. Nele, novamente, criamos uma semente randomicamente, e através da inversa da fórmula que nos é passado no PDF:

$$V(u) = \left\lceil \frac{\ln(1-u)}{\ln(1-\frac{1}{22})} \right\rceil$$

Conseguimos calcular a média (usando a fórmula da média nessa função). Como resultado do teste recebemos:

Sample Mean: 21.9926

Como a média apresentada no PDF é 22, logo o resultado se apresenta próximo suficiente.

2.4 event list test

```
// Tests if the insertion and removal in the list are right
void event_list_test(){
    list<Event> event_list;
    char input;
    double input2;
    int input3;
    int id = 0;

    cout << "COMMANDS:\n\tP = Print event list\n\tI = Insert event\n\tR = Remove event\n\tE or Q = Exit test\n\n";

    while(input != 'e' && input != 'E' && input != 'q' && input != 'Q') {
        cout << "\n";
        cin >> input;
        if (input == 'p' || input == 'P') {
            cout << "\nEvent list:";
            for (list<Event>::iterator it = event_list.begin(); it != event_list.end(); it++)
                cout << ' ' << it->time << "(" << it->customer.id << ")";
            cout << endl;
        } else if (input == 'i' || input == 'I') {
            cout << "Time: ";
            cin >> input2;
            list_insert(event_list, Event(input2, Customer(id++, 0, DATA, input2), ARRIVAL));
        } else if (input == 'r' || input == 'R') {
            cout << "ID: ";
            cin >> input3;
            list_remove(event_list, input3);
        }

        cout << "COMMANDS:\n\tP = Print event list\n\tI = Insert event\n\tR = Remove event\n\tE or Q = Exit test\n\n";
    }
    cout << "\n";
}
```

Esse teste foi feito para verificar a implementação da fila em ordem cronológica. Nesse teste é possível inserir, printar (para verificar se a ordem está correta), remover e sair do próprio teste.

Para realizar o teste foram inseridos os valores 5, 2, 7 e 1, na respectiva ordem. Depois de suas inserções, foi printada a lista correspondente (ao lado do número aparece o seu respectivo ID, que é usado na remoção). Para confirmar a remoção, foi removido, após, o número 5 e printada novamente a lista, tendo tido esses resultados, confirmado que a inserção e remoção da lista estavam implementadas de maneira correta:

- Primeira vez: Event list: 1(3) 2(1) 5(0) 7(2)
- Segunda vez: Event list: 1(3) 2(1) 7(2)

2.5 queue test

```
// Tests the queue about receiving new packages
void queue_test(){
    queue *data_queue = queue_create();
    queue *voice_queue = queue_create();
    char input = '\0';
    int timestamp = 0;

    cout << "COMMANDS:\n\tD = New data package\n\tV = New voice package\n\tP = Print next\n\tE or Q = Exit test\n\n";

    while (input != 'e' && input != 'E' && input != 'q' && input != 'Q') {
        cout << "Data packages: " << data_queue->size << "\n";
        cout << "Voice packages: " << voice_queue->size << "\n";
        cout << "\n";
        cin >> input;
        if (input == 'D' || input == 'd') {
            Customer c = Customer(0, 0, DATA, timestamp++);
            queue_insert(data_queue, c);
            cout << "Inserted data package! (size = " << c.size << " bits)\n";
        } else if (input == 'V' || input == 'v') {
            Customer c = Customer(0, 0, VOICE, timestamp++);
            queue_insert(voice_queue, c);
            cout << "Inserted voice package! (size = " << c.size << " bits)\n";
        } else if (input == 'P' || input == 'p') {
            if (voice_queue->size > 0) {
                Customer c = queue_remove(voice_queue);
                cout << "Type: " << c.type << "\nSize: " << c.size << "\n\n";
            }
            else if (data_queue->size > 0) {
                Customer c = queue_remove(data_queue);
                cout << "Type: " << c.type << "\nSize: " << c.size << "\n\n";
            }
        }
        cout << "COMMANDS:\n\tD = New data package\n\tV = New voice package\n\tP = Print next\n\tE or Q = Exit test\n\n";
    }

    cout << '\n';
}
```

Essa implementação é o teste das filas, onde temos a entrada de um pacote de voz e de dados. No teste, podemos inserir um pacote de voz na fila das vozes e um pacote de dados na fila dos dados. Caso seja o print (se assemelha

a remoção de um pacote) da fila, mostramos quem seria atendido primeiro de ambas as filas, e como voz tem prioridade sobre dado, se a fila das vozes não estiver fazia, fazemos com que o fregues a ser atendido seja o primeiro de tal fila, mas se estiver, pegamos o fregues da fila dos dados (caso tenha).

Na execução do teste, foi inserido um dado e printada a fila, que aponta para o dado. Logo depois, foi inserido um dado e uma voz e printada a fila, que agora aponta para a voz, já que ela possui prioridade.

- Primeira vez: Type: 1 - Size: 512
- Segunda vez: Type: 2 - Size: 512

2.6 round test

Esse é o último teste que foi implementado, e serve para testar o funcionamento da função principal do programa. Ele é basicamente a mesma função principal, mas possui a especificidade de ter o período de silêncio determinístico para cada canal como $100 + (\text{channel_id} * 5)$ e o tempo de chegada de um pacote de dados também determinístico como 60.

Nessa função, foram feitos dois testes, sem pacote de dados e com pacote de dados, e cada arquivo log (para saber o que foi executado) é gravado em um arquivo diferente.

O código é:

```
// Creates the arrival event of a data customer with deterministic time
Event create_data_deterministic(int customer_id, int round, float simulation_time,
float lambda){
double arrivalTime = 60;
Customer customer = Customer(customer_id, round, DATA, simulation_time + arrivalTime);
Event event = Event(simulation_time + arrivalTime, customer, ARRIVAL);
return event;
}

// Creates the silence period event with deterministic time
Event create_silence_period_deterministic(float simulation_time, float offset,
int channel_id){
double arrivalTime = 100 + (channel_id * 5);
Event event = Event(simulation_time + arrivalTime + offset, channel_id);
return event;
}

// Creates the silence period event with exponential time
Event create_silence_period_random(float simulation_time, float offset, int channel_id){
double arrivalTime = exponential(1.0/650);
Event event = Event(simulation_time + arrivalTime + offset, channel_id);
```

```

    return event;
}

// It runs the rounds of the simulation for deterministic silence period if has_data
is false and for deterministic data otherwise
void round_test(int transiente_period, int customers_number, int round_number,
float lambda, bool preemption, bool allow_logging, bool has_data){
queue* data_traffic = queue_create(); // Queue where the data packages are stored
queue* voice_traffic = queue_create(); // Queue where the voice packages are stored

double simulation_time = 0; // Current time in the simulator
double last_time[30]; //Keeps the last time a package of each specific channel
entered the server

for (int i = 0; i < 30; i++){
last_time[i] = 0;
}

// Gets the intervals of each channel;
vector <double> intervals;

// Since we must ignore the first transiente_period customers, only customers
considered will be the ones with id >= 0
// and we'll start counting from -transiente_period
Customer::total_customers = -transiente_period;

Customer customer_being_served = Customer(); // The customer currently in the server. NONE
type = no customer there.

// Expectations/Averages
float T1[round_number+1]; // Average time a data package stays in the system
float W1[round_number+1]; // Average time a data package stays in the queue
float X1[round_number+1]; // Average time a data package stays in the server
    float Nq1[round_number+1]; // Average number of data packages in the queue
float T2[round_number+1]; // Average time a voice package stays in the system
float W2[round_number+1]; // Average time a voice package stays in the queue
    float Nq2[round_number+1]; // Average number of voice packages in the queue

// Interval between packages Estimator
    float EDelta[round_number+1];
    float VDelta[round_number+1];

for (int i = 0; i < round_number+1; i++) {
T1[i] = 0;
W1[i] = 0;
X1[i] = 0;
}

```

```

Nq1[i] = 0;
T2[i] = 0;
W2[i] = 0;
Nq2[i] = 0;

EDelta[i] = 0;
VDelta[i] = 0;
}

// Averages, lower limits and upper limits of the confidence intervals
float ET1 = 0, lower_T1, upper_T1;
float EW1 = 0, lower_W1, upper_W1;
float EX1 = 0, lower_X1, upper_X1;
float ENq1 = 0, lower_Nq1, upper_Nq1;
float ET2 = 0, lower_T2, upper_T2;
float EW2 = 0, lower_W2, upper_W2;
float ENq2 = 0, lower_Nq2, upper_Nq2;
float EEDelta = 0, lower_EDelta, upper_EDelta;
float EVDelta = 0, lower_VDelta, upper_VDelta;

// Standard deviations of the confidence intervals
float ST1 = 0;
float SW1 = 0;
float SX1 = 0;
float SNq1 = 0;
float ST2 = 0;
float SW2 = 0;
float SNq2 = 0;
float SEDelta = 0;
float SVDelta = 0;

// Variables used for the areas method (Data Queue)
double time_data = 0; // data queue timestamps
int size_data; // data queue sizes

// Variables used for the areas method (Voice Queue)
double time_voice = 0; // voice queue timestamps
int size_voice; // voice queue sizes

// People that came out of the system coming from both Queues;
int out1 = 0; // data packages
int out2 = 0; // voice packages
int out = 0; // out1 + out2

// How many voice packages each channel needs to send before entering silence period.
int voice_channels[30];

```

```

for (int i = 0; i < 30; i++) voice_channels[i] = 0;

list<Event> event_list;

// If this is the deterministic data test, than the data package will be created
if (has_data) list_insert(event_list, create_data_deterministic(
Customer::total_customers++, 0, simulation_time, lambda));

// VOICE CHANNELS
for(int i = 0; i < 30; i++) {
    if (!has_data) list_insert(event_list, create_silence_period_deterministic(
simulation_time, 0, i));
    else list_insert(event_list, create_silence_period_random(simulation_time, 0, i));
}

// Number of packages created in each round that eventually left the system
int round_data_exits[round_number];
for (int i = 0; i < round_number; i++) round_data_exits[i] = 0;
int round_voice_exits[round_number];
for (int i = 0; i < round_number; i++) round_voice_exits[i] = 0;

//DEBUG FILES
ofstream log_file, averages_file;
if (allow_logging) {
// Determines each file for both tests
if (!has_data) log_file.open ("log_deterministic_test.txt");
else log_file.open ("log_deterministic_data_test.txt");
}

int simulation_percentage = -1;
cout << endl;

cout << "Simulation in progress... 0%\r";

// Main loop of events
for (int round = 0; round < round_number+1; round++) {
//--- Variables needed to calculate the statistics for each round ---//
double round_time = simulation_time;
//-----// 

if (allow_logging) {
stringstream sstm;
sstm << "log_" << round << ".txt";
log_file.open (sstm.str().c_str());
}
// Loops until customers_number customers are sampled or, if this is round 0, until

```

```

the transiente_period customers are sampled
while ((round > 0 && Customer::total_customers < customers_number * (round)) ||
(round == 0 && Customer::total_customers < 0)) {
Event current_event = *event_list.begin();

event_list.erase(event_list.begin());
Customer c_prev = customer_being_served; // needed to test if "treat_event" will
change the customer in the server
int data_queue_prev = data_traffic->size; // needed to test if "treat_event" will
interrupt a data package being served
int voice_queue_prev = voice_traffic->size; // both of these are needed for the Areas
Method
current_event.treat_event(data_traffic, voice_traffic, &customer_being_served, preemption);
simulation_time = current_event.time;

//===== Areas Method =====//
if (data_queue_prev != data_traffic->size) {
if (time_data == 0) {
time_data = simulation_time;
size_data = data_traffic->size;
} else {
Nq1[round] += fabs((simulation_time - time_data) * (size_data));
time_data = simulation_time;
size_data = data_traffic->size;
}
}
if (voice_queue_prev != voice_traffic->size) {
if (time_voice == 0) {
time_voice = simulation_time;
size_voice = voice_traffic->size;
} else {
Nq2[round] += fabs((simulation_time - time_voice) * (size_voice));
time_voice = simulation_time;
size_voice = voice_traffic->size;
}
}
//=====//

if (current_event.etype == ARRIVAL && current_event.customer.type == DATA) {
list_insert(event_list, create_data_deterministic(-99999, round, simulation_time, lambda));
// next data package
} else if (current_event.etype == ARRIVAL && current_event.customer.type == VOICE){
if (voice_channels[current_event.channel_id] > 0) {
voice_channels[current_event.channel_id]--;

```

```

list_insert(event_list, create_voice(-99999, round, simulation_time,
16, current_event.channel_id)); // next voice package of this channel
} else {
if (!has_data) list_insert(event_list, create_silence_period_deterministic(
simulation_time, 16, current_event.channel_id)); // starts next silence period 16ms later
else list_insert(event_list, create_silence_period_random(
simulation_time, 16, current_event.channel_id));
}
if (data_queue_prev < data_traffic->size) { // if a voice arrival increased the data queue,
that means a data package was interrupted
Customer removed_customer = queue_remove(data_traffic); // the data package that
suffered interruption
removed_customer.time_in_service += (simulation_time - removed_customer.checkpoint_time);
removed_customer.checkpoint_time = simulation_time;
queue_return(data_traffic, removed_customer);
if (allow_logging) log_file << "1 -- " << "REMOVED " << removed_customer.id <<
" FROM DATA QUEUE" << endl;
list_remove(event_list, removed_customer.id);
}

} else if (current_event.etype == SILENCE_END) {
if (last_time[current_event.channel_id] != 0) last_time[current_event.channel_id] = 0;
voice_channels[current_event.channel_id] = voice_package_number();
if (voice_channels[current_event.channel_id] > 0) {
voice_channels[current_event.channel_id]--;
list_insert(event_list, create_voice(-99999, round, simulation_time, 0,
current_event.channel_id)); // next voice package of this channel
}
} else if (current_event.etype == EXIT && current_event.customer.type == DATA){
T1[current_event.customer.round] += (current_event.time -
current_event.customer.arrival_time);
W1[current_event.customer.round] += current_event.customer.time_in_queue;
current_event.customer.time_in_service += (simulation_time -
current_event.customer.checkpoint_time);
X1[current_event.customer.round] += current_event.customer.time_in_service;

round_data_exits[current_event.customer.round]++;
} else if (current_event.etype == EXIT && current_event.customer.type == VOICE){
T2[current_event.customer.round] += (current_event.time -
current_event.customer.arrival_time);
W2[current_event.customer.round] += current_event.customer.time_in_queue;

round_voice_exits[current_event.customer.round]++;
}
if (customer_being_served.id != c_prev.id) { // checks if a new customer arrived
at the server due to this event

```

```

if (customer_being_served.type != NONE) {
    customer_being_served.time_in_queue += (simulation_time -
    customer_being_served.checkpoint_time);
    customer_being_served.checkpoint_time = simulation_time;
    //If a Voice is being served, gets the interval
    if(customer_being_served.type == VOICE){
        //If it is not the first voice package of the channel
        if (last_time[customer_being_served.channel_id] != 0){
            intervals.push_back(customer_being_served.checkpoint_time -
            last_time[customer_being_served.channel_id]);
        }
        last_time[customer_being_served.channel_id] = customer_being_served.checkpoint_time;
    }
    list_insert(event_list, remove_package(simulation_time, customer_being_served));
}
}

// Percentage of simulation complete
if (Customer::total_customers > 0 && simulation_percentage != (int)((float)
Customer::total_customers * 100 / (customers_number * round_number))) {
    cout << "Simulation in progress... " << (int)((float) Customer::total_customers * 100
    / (customers_number * round_number)) << "%" << '\r';
    simulation_percentage = (int)((float) Customer::total_customers * 100 /
    (customers_number * round_number));
}

//=====
if (allow_logging) {
    log_file << "1 -- " << current_event.time << "ms TYPE: ";
    if (current_event.etype == ARRIVAL && current_event.customer.type == DATA) log_file <<
    "Data Arrival";
    else if (current_event.etype == ARRIVAL && current_event.customer.type == VOICE) log_file <<
    "Voice Arrival";
    else if (current_event.etype == SILENCE_END) log_file << "Silence End";
    else if (current_event.etype == EXIT && current_event.customer.type == DATA) log_file <<
    "Data Departure";
    else if (current_event.etype == EXIT && current_event.customer.type == VOICE) log_file <<
    "Voice Departure";
    if (current_event.etype != SILENCE_END) log_file << " CUSTOMER ID: "<<
    (current_event.customer.id);
    log_file << " CHANNEL ID: " << current_event.channel_id << "\n";
}
//=====

}

```

```

if (allow_logging) {
    log_file << "End of Round " << round << " ; Duration: " << (simulation_time - round_time)
    << "ms\n\n";
    log_file.close();
}
// Areas Method requires dividing the area by the time spent
Nq1[round] /= (simulation_time - round_time);
Nq2[round] /= (simulation_time - round_time);

// Sum all the intervals into delta of the round
for (int i = 0; i < intervals.size(); i++){
    EDelta[round] += intervals[i];
}

// If there is some interval, divide by the quantity to get the average
if (!intervals.empty()) EDelta[round] = EDelta[round]/intervals.size();

for (int i = 0; i < intervals.size(); i++){
    VDelta[round] += pow((intervals[i] - EDelta[round]), 2);
}

if (intervals.size() <= 1) VDelta[round] = 0;
else VDelta[round] = VDelta[round]/(intervals.size() - 1);

intervals.clear();
}

// Divide the sum of times by the number of events to find the average
for (int round = 0; round < round_number+1; round++) {
    if (round_data_exits[round] > 0) {
        T1[round] /= round_data_exits[round];
        W1[round] /= round_data_exits[round];
        X1[round] /= round_data_exits[round];
    }
    if (round_voice_exits[round] > 0) {
        T2[round] /= round_voice_exits[round];
        W2[round] /= round_voice_exits[round];
    }
}

cout << endl;

// ----- Finding the averages of the confidence intervals ----- //
// Only rounds from which at least one package left the system are eligible for certain

```

```

statistics.

int eligible_data_rounds = 0, eligible_voice_rounds = 0;
for(int i=1; i < round_number+1; i++) {
    if (round_data_exits[i] > 0) eligible_data_rounds++;
    if (round_voice_exits[i] > 0) eligible_voice_rounds++;
    ET1 += T1[i];
    EW1 += W1[i];
    EX1 += X1[i];
    ENq1 += Nq1[i];
    ET2 += T2[i];
    EW2 += W2[i];
    ENq2 += Nq2[i];
    EEDelta += EDelta[i];
    EVDelta += VDelta[i];
}
ET1 /= eligible_data_rounds;
EW1 /= eligible_data_rounds;
EX1 /= eligible_data_rounds;
ENq1 /= round_number;
ET2 /= eligible_voice_rounds;
EW2 /= eligible_voice_rounds;
ENq2 /= round_number;
EEDelta /= round_number;
EVDelta /= round_number;

// Finding the standard deviations of the confidence intervals
for(int i=1; i < round_number+1; i++) {
    ST1 += pow(T1[i] - ET1, 2);
    SW1 += pow(W1[i] - EW1, 2);
    SX1 += pow(X1[i] - EX1, 2);
    SNq1 += pow(Nq1[i] - ENq1, 2);
    ST2 += pow(T2[i] - ET2, 2);
    SW2 += pow(W2[i] - EW2, 2);
    SNq2 += pow(Nq2[i] - ENq2, 2);
    SEDelta += pow(EDelta[i] - EEDelta, 2);
    SVDelta += pow(VDelta[i] - EVDelta, 2);
}
ST1 /= (round_number - 1);
ST1 = sqrt(ST1);
SW1 /= (round_number - 1);
SW1 = sqrt(SW1);
SX1 /= (round_number - 1);
SX1 = sqrt(SX1);
SNq1 /= (round_number - 1);
SNq1 = sqrt(SNq1);
ST2 /= (round_number - 1);

```

```

ST2 = sqrt(ST2);
SW2 /= (round_number - 1);
SW2 = sqrt(SW2);
SNq2 /= (round_number - 1);
SNq2 = sqrt(SNq2);
SEDelta /= (round_number - 1);
SEDelta = sqrt(SEDelta);
SVDelta /= (round_number - 1);
SVDelta = sqrt(SVDelta);

// Finding the confidence intervals.
// Lower limits must always be non-negative.
lower_T1 = ET1 - 1.645 * ST1 / sqrt(round_number); lower_T1 = (lower_T1 > 0) ?
lower_T1 : 0; upper_T1 = ET1 + 1.645 * ST1 / sqrt(round_number);
lower_W1 = EW1 - 1.645 * SW1 / sqrt(round_number); lower_W1 = (lower_W1 > 0) ?
lower_W1 : 0; upper_W1 = EW1 + 1.645 * SW1 / sqrt(round_number);
lower_X1 = EX1 - 1.645 * SX1 / sqrt(round_number); lower_X1 = (lower_X1 > 0) ?
lower_X1 : 0; upper_X1 = EX1 + 1.645 * SX1 / sqrt(round_number);
lower_Nq1 = ENq1 - 1.645 * SNq1 / sqrt(round_number); lower_Nq1 = (lower_Nq1 > 0) ?
lower_Nq1 : 0; upper_Nq1 = ENq1 + 1.645 * SNq1 / sqrt(round_number);
lower_T2 = ET2 - 1.645 * ST2 / sqrt(round_number); lower_T2 = (lower_T2 > 0) ?
lower_T2 : 0; upper_T2 = ET2 + 1.645 * ST2 / sqrt(round_number);
lower_W2 = EW2 - 1.645 * SW2 / sqrt(round_number); lower_W2 = (lower_W2 > 0) ?
lower_W2 : 0; upper_W2 = EW2 + 1.645 * SW2 / sqrt(round_number);
lower_Nq2 = ENq2 - 1.645 * SNq2 / sqrt(round_number); lower_Nq2 = (lower_Nq2 > 0) ?
lower_Nq2 : 0; upper_Nq2 = ENq2 + 1.645 * SNq2 / sqrt(round_number);
lower_EDelta = EEDelta - 1.645 * SEDelta / sqrt(round_number); lower_EDelta =
(lower_EDelta > 0) ? lower_EDelta : 0; upper_EDelta = EEDelta + 1.645 * SEDelta /
sqrt(round_number);
lower_VDelta = EVDelta - 1.645 * SVDelta / sqrt(round_number); lower_VDelta =
(lower_VDelta > 0) ? lower_VDelta : 0; upper_VDelta = EVDelta + 1.645 * SVDelta /
sqrt(round_number);

// Creates a file where all the averages of each round are stored
if (allow_logging) {
if (!has_data) averages_file.open ("averages_deterministic_test.txt");
else {
averages_file.open ("averages_deterministic_data_test.txt");
averages_file << "\nT1: [ ";
for (int i = 0; i < round_number-1; i++) averages_file << T1[i] << ", ";
averages_file << T1[round_number-1] << " ]\n";

averages_file << "\nW1: [ ";
for (int i = 0; i < round_number-1; i++) averages_file << W1[i] << ", ";
averages_file << W1[round_number-1] << " ]\n";
}
}

```

```

averages_file << "\nX1: [ ";
for (int i = 0; i < round_number-1; i++) averages_file << X1[i] << ", ";
averages_file << X1[round_number-1] << " ]\n";

averages_file << "\nNq1: [ ";
for (int i = 0; i < round_number-1; i++) averages_file << Nq1[i] << ", ";
averages_file << Nq1[round_number-1] << " ]\n";
}

averages_file << "\nT2: [ ";
for (int i = 0; i < round_number-1; i++) averages_file << T2[i] << ", ";
averages_file << T2[round_number-1] << " ]\n";

averages_file << "\nW2: [ ";
for (int i = 0; i < round_number-1; i++) averages_file << W2[i] << ", ";
averages_file << W2[round_number-1] << " ]\n";

averages_file << "\nNq2: [ ";
for (int i = 0; i < round_number-1; i++) averages_file << Nq2[i] << ", ";
averages_file << Nq2[round_number-1] << " ]\n";

averages_file << "\nE[Delta]: [ ";
for (int i = 0; i < round_number-1; i++) averages_file << EDelta[i] << ", ";
averages_file << EDelta[round_number-1] << " ]\n";

averages_file << "\nV[Delta]: [ ";
for (int i = 0; i < round_number-1; i++) averages_file << VDelta[i] << ", ";
averages_file << VDelta[round_number-1] << " ]\n";

averages_file.close();
}

// Prints the results
cout << "Intervalos de confianca: " << endl;
if (has_data){
    cout << "\nE[T1]: " << lower_T1 << " < " << ET1 << " < " << upper_T1 << endl;
    cout << "\nE[W1]: " << lower_W1 << " < " << EW1 << " < " << upper_W1 << endl;
    cout << "\nE[X1]: " << lower_X1 << " < " << EX1 << " < " << upper_X1 << endl;
    cout << "\nE[Nq1]: " << lower_Nq1 << " < " << ENq1 << " < " << upper_Nq1 << endl;
}
cout << "\nE[T2]: " << lower_T2 << " < " << ET2 << " < " << upper_T2 << endl;
cout << "\nE[W2]: " << lower_W2 << " < " << EW2 << " < " << upper_W2 << endl;
cout << "\nE[Nq2]: " << lower_Nq2 << " < " << ENq2 << " < " << upper_Nq2 << endl;
cout << "\nE[Delta]: " << lower_EDelta << " < " << EEDelta << " < " <<
upper_EDelta << endl;
cout << "\nV[Delta]: " << lower_VDelta << " < " << EVDelta << " < " <<

```

```

    upper_VDelta << endl;
}


```

Através dele é possível testar os casos em que temos o período de silêncio determinístico com e sem pacotes de dados determinísticos.

3 Estimativa de Fase Transiente

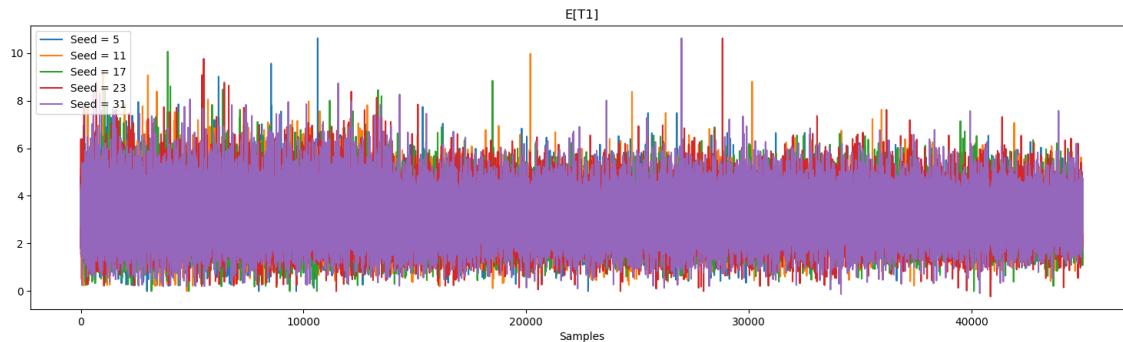
Para estimarmos a fase transiente, foram traçados diversos gráficos utilizando a biblioteca Matplotlib do Python. Cada gráfico diz respeito à esperança de uma das sete estatísticas base, para cada um dos sete valores possíveis de ρ , para cada uma das duas disciplinas (não-preemptiva e preemptiva). Para garantir que cada gráfico fosse confiável, cada um deles apresenta, simultaneamente, cinco diferentes rodadas, cada uma com uma semente diferente.

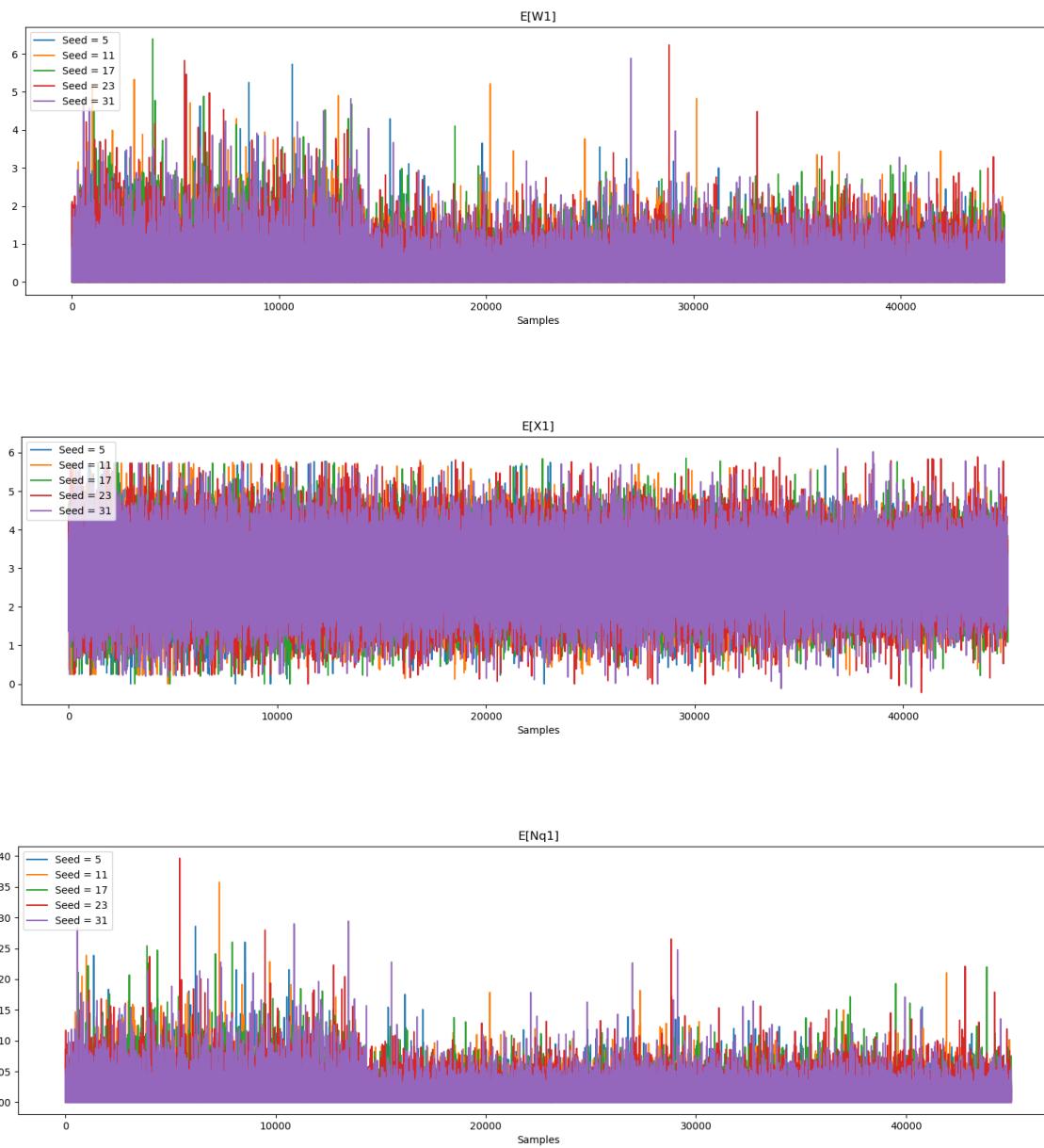
A escolha das sementes foi feita, como visto em aula, escolhendo números primos relativamente distantes entre si, o que garante uma maior variação nas amostragens.

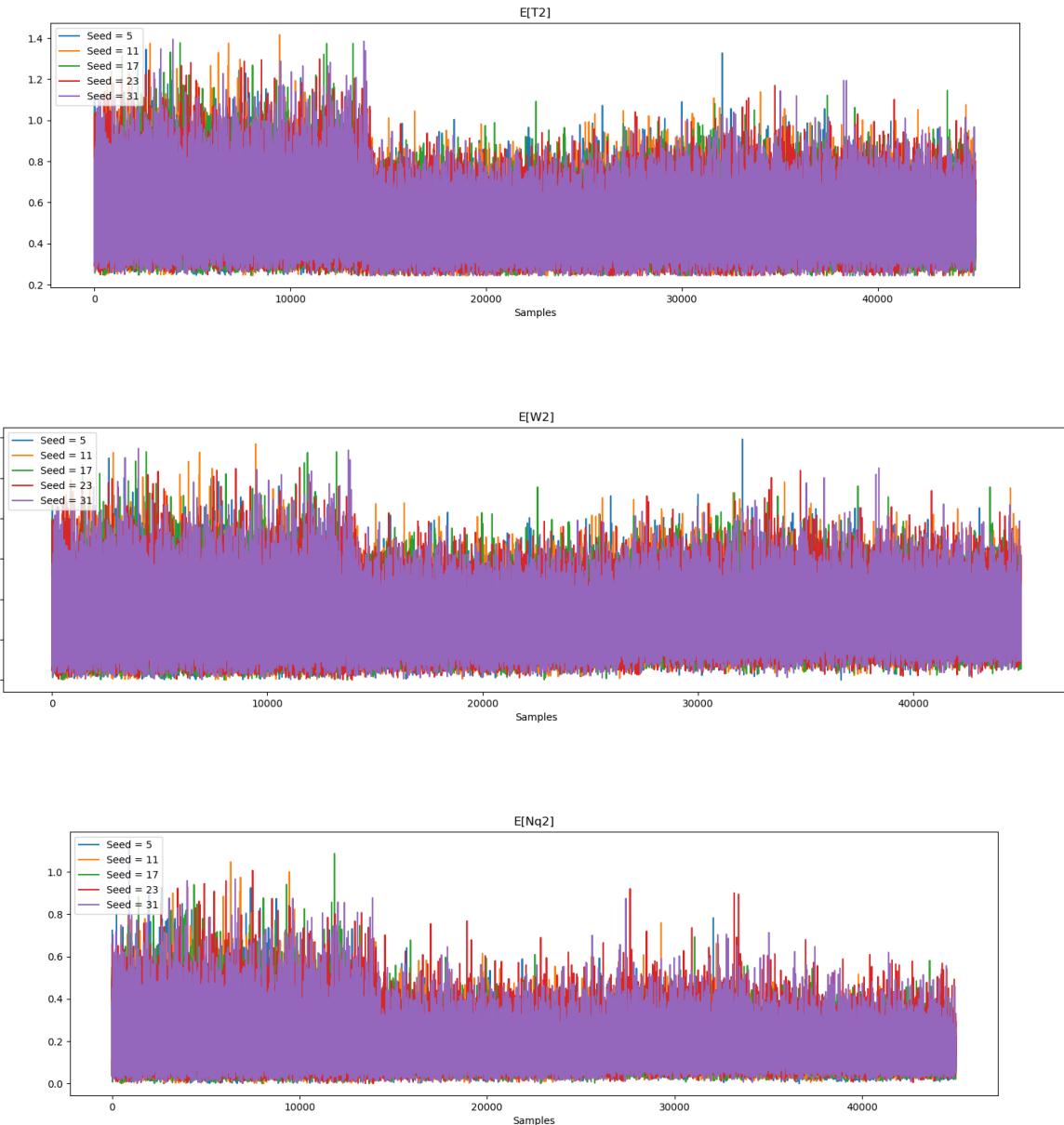
Portanto, cada rodada do simulador para os seguintes gráficos foi feita utilizando as sementes 5, 11, 17, 23 e 31. O número de fregueses por rodada escolhido foi 200, enquanto que o número de rodadas foi 45000. Isso garante que teremos um número muito alto de fregueses amostrados, bem como um número muito alto de médias a serem observadas.

Nota: Alguns gráficos de $E[T_1]$ e $E[W_1]$ com disciplina preemptiva possuem quedas abruptas para zero. Isso se dá pois, da forma como o simulador foi implementado, pacotes de dados só contribuem para as estatísticas de tempo quando saem do sistema. Isso impede que tempos de espera parciais (menores) influenciem no resultado. As rodadas em que nenhum pacote saiu do sistema (e, portanto, nenhum pacote contribuiu com o cálculo dessas estatísticas) são desprezadas no programa, apesar de serem representadas nos gráficos.

3.1 Para $\rho=0.1$ (não-preemptivo)



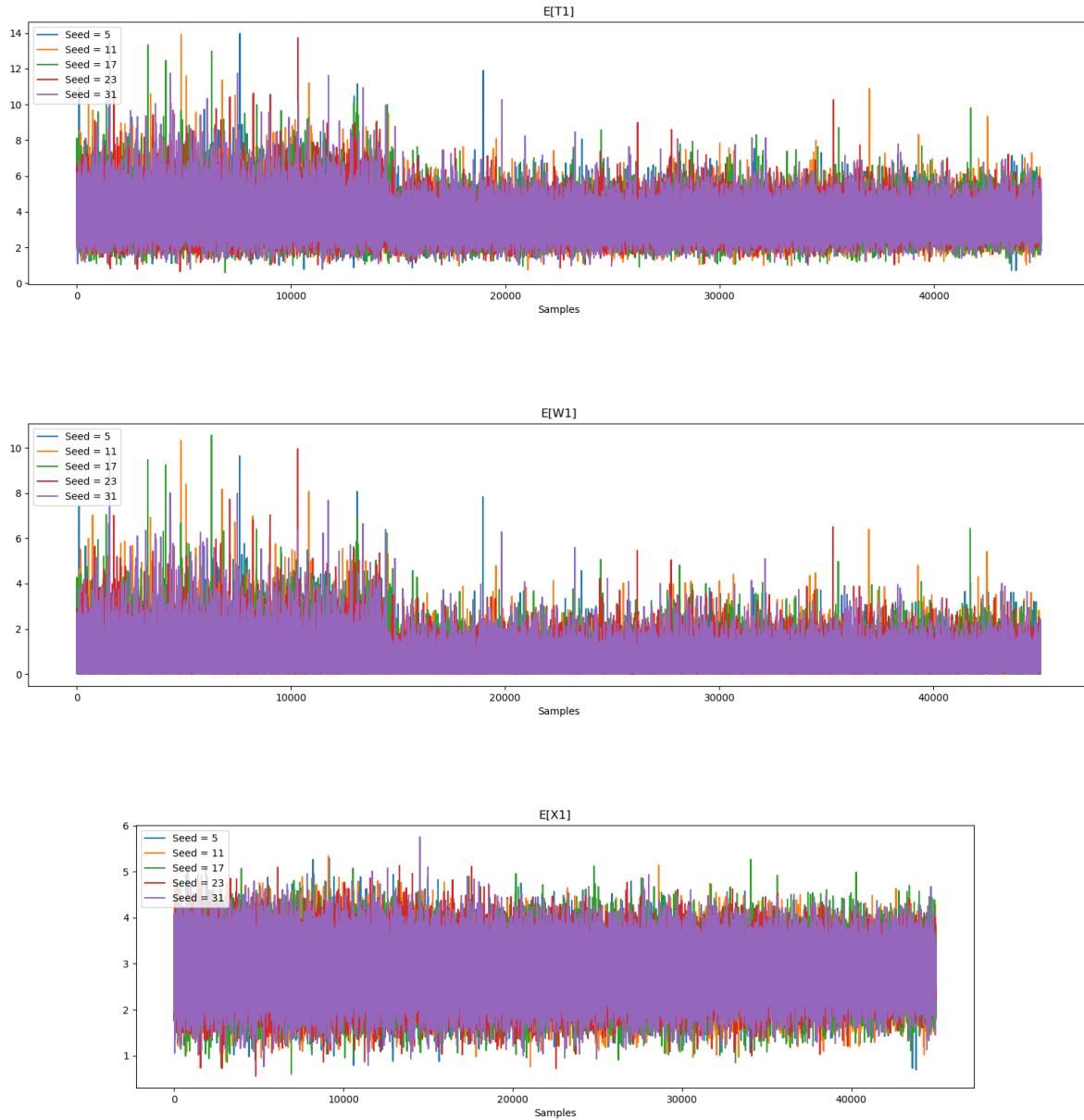


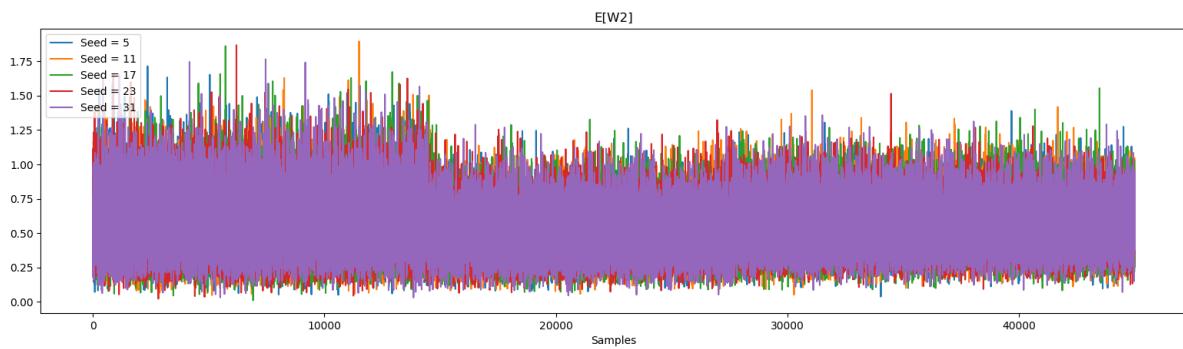
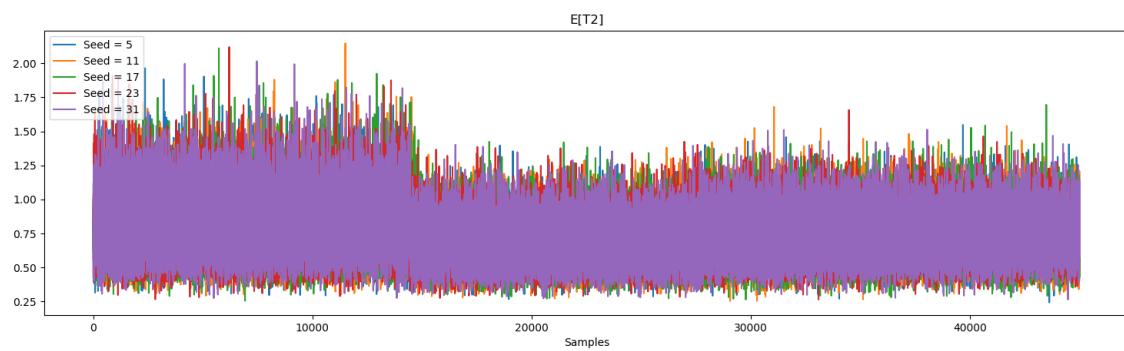
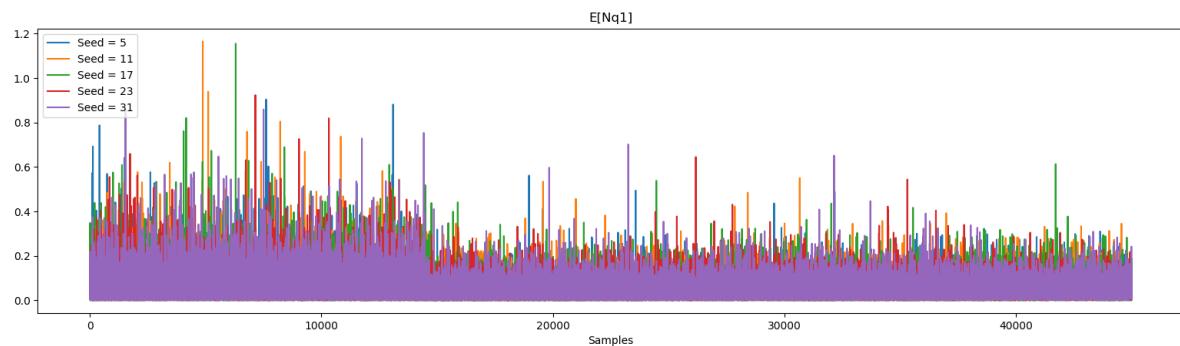


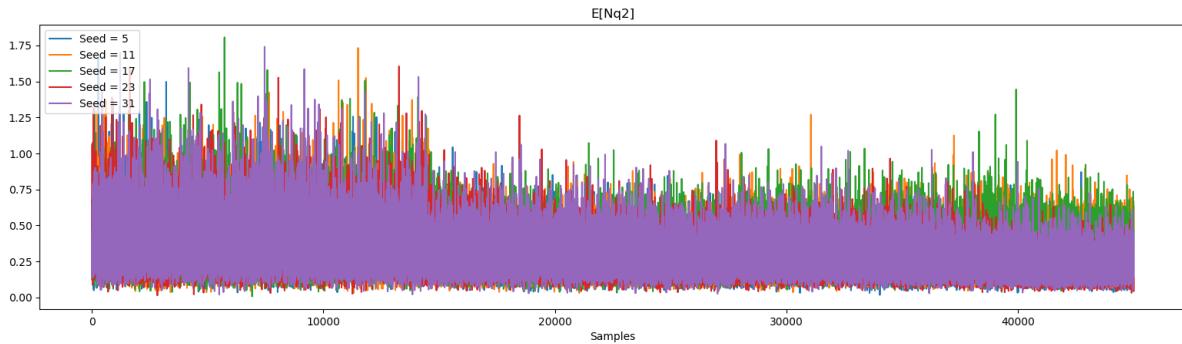
Podemos ver que os gráficos que melhor indicam uma convergência nos valores são os gráficos de $E[T1]$ e $E[T2]$. Esses gráficos indicam um equilíbrio a partir de, aproximadamente, 14000 rodadas, o que significam **2800000** amostras. Isso indica que, para alcançar um sistema em equilíbrio e otimizar o encontro de um intervalo de confiança pequeno, seria necessário ignorar um número muito

grande de amostras. Por outro lado, qualquer número abaixo de 2800000 pode ser usado como fase transitória a fim de diminuir o número total de amostras do simulador.

3.2 Para $\rho=0.2$ (não-preemptivo)

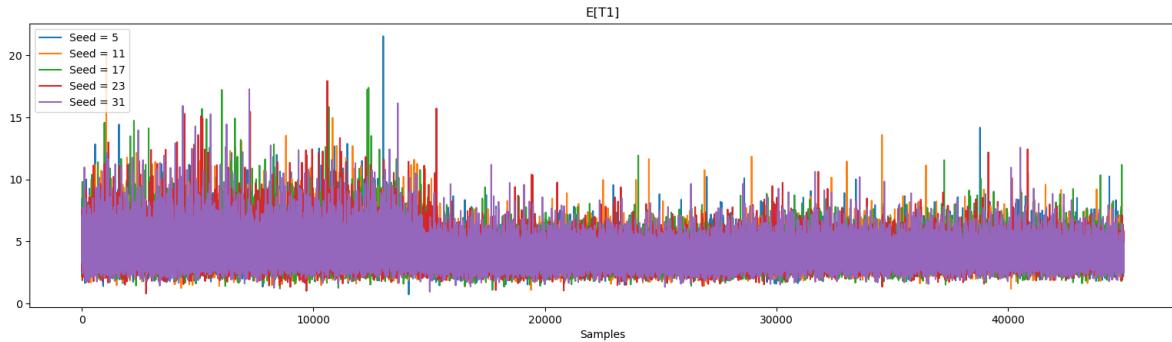


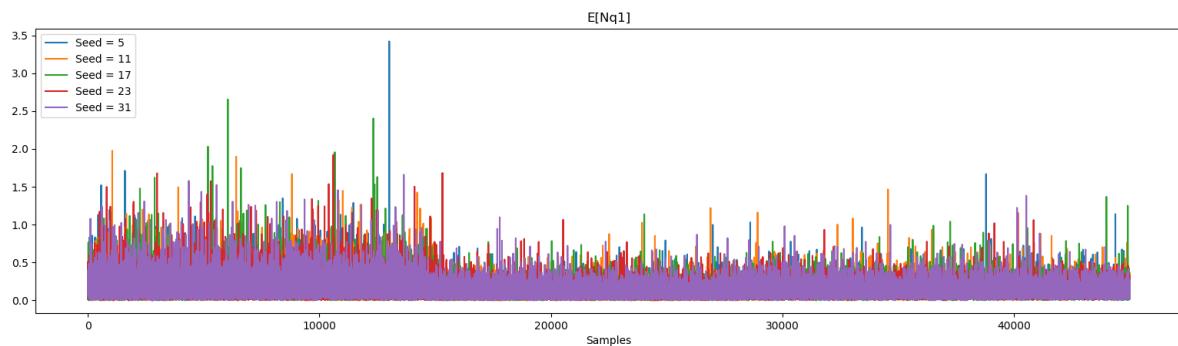
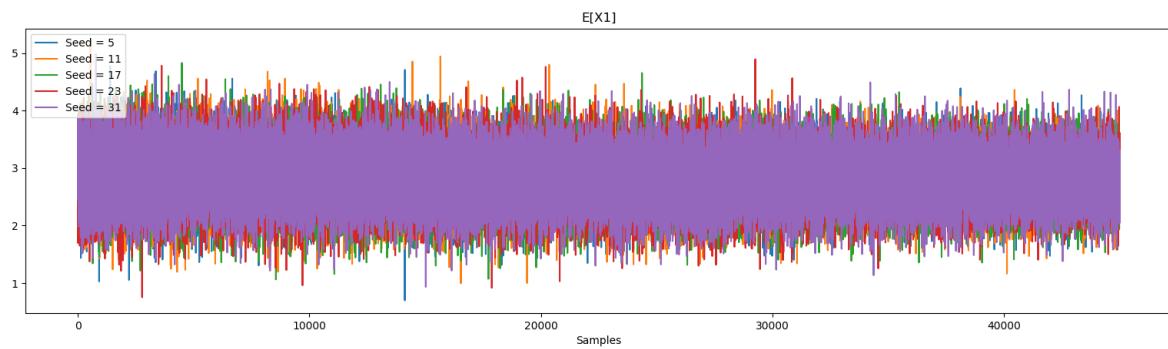
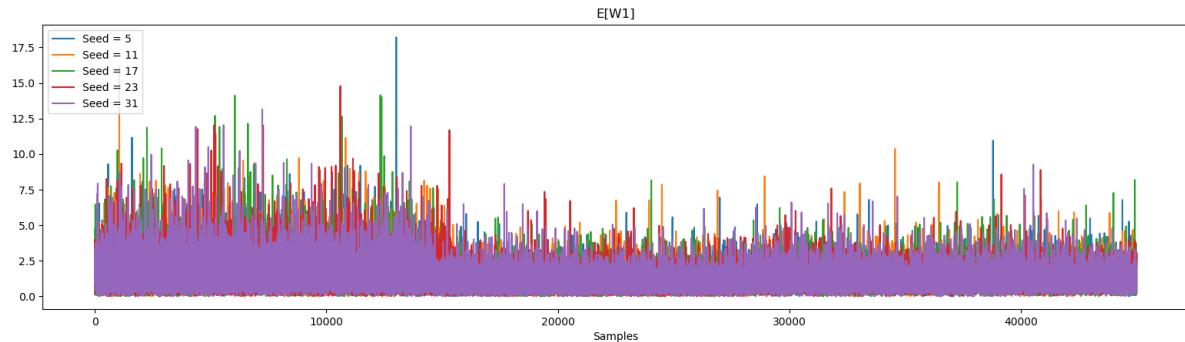


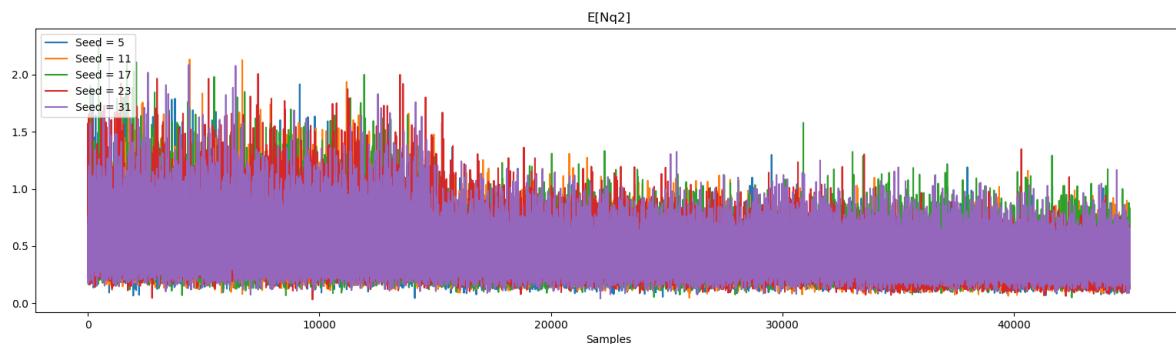
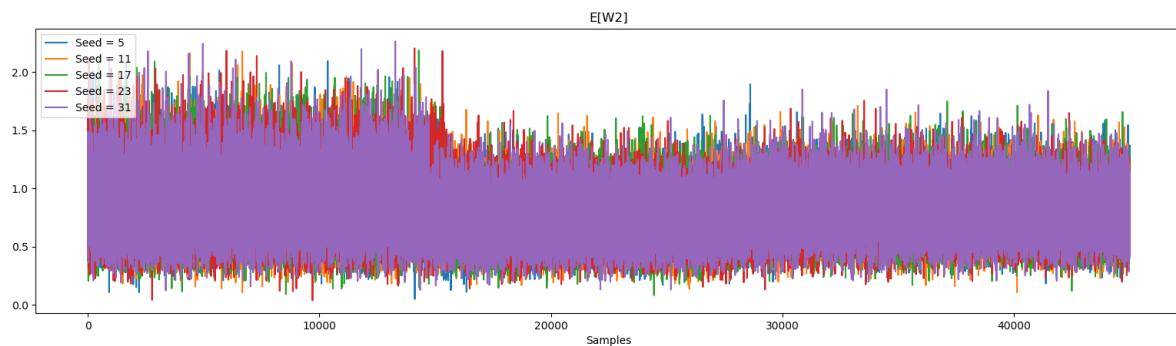
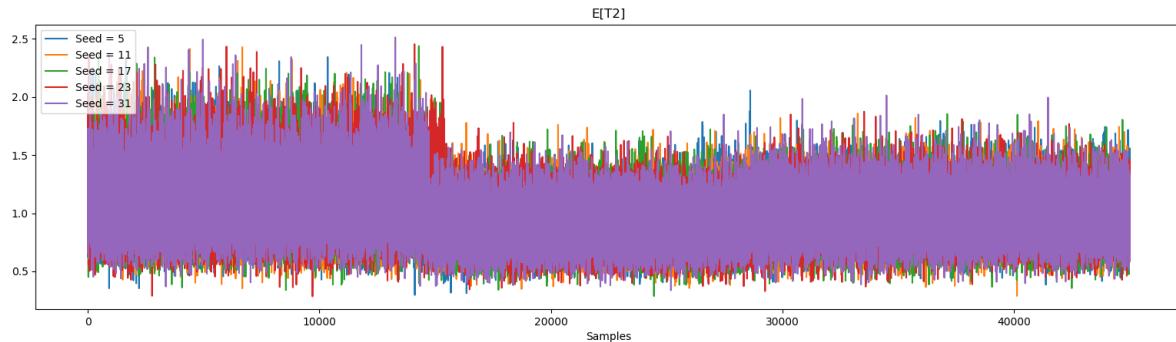


Podemos ver que os gráficos que melhor indicam uma convergência nos valores são os gráficos de $E[T1]$ e $E[T2]$. Esses gráficos indicam um equilíbrio a partir de, aproximadamente, 14000 rodadas, o que significam **2800000** amostras. Isso indica que, para alcançar um sistema em equilíbrio e otimizar o encontro de um intervalo de confiança pequeno, seria necessário ignorar um número muito grande de amostras. Por outro lado, qualquer número abaixo de 2800000 pode ser usado como fase transitória a fim de diminuir o número total de amostras do simulador.

3.3 Para $\rho=0.3$ (não-preemptivo)



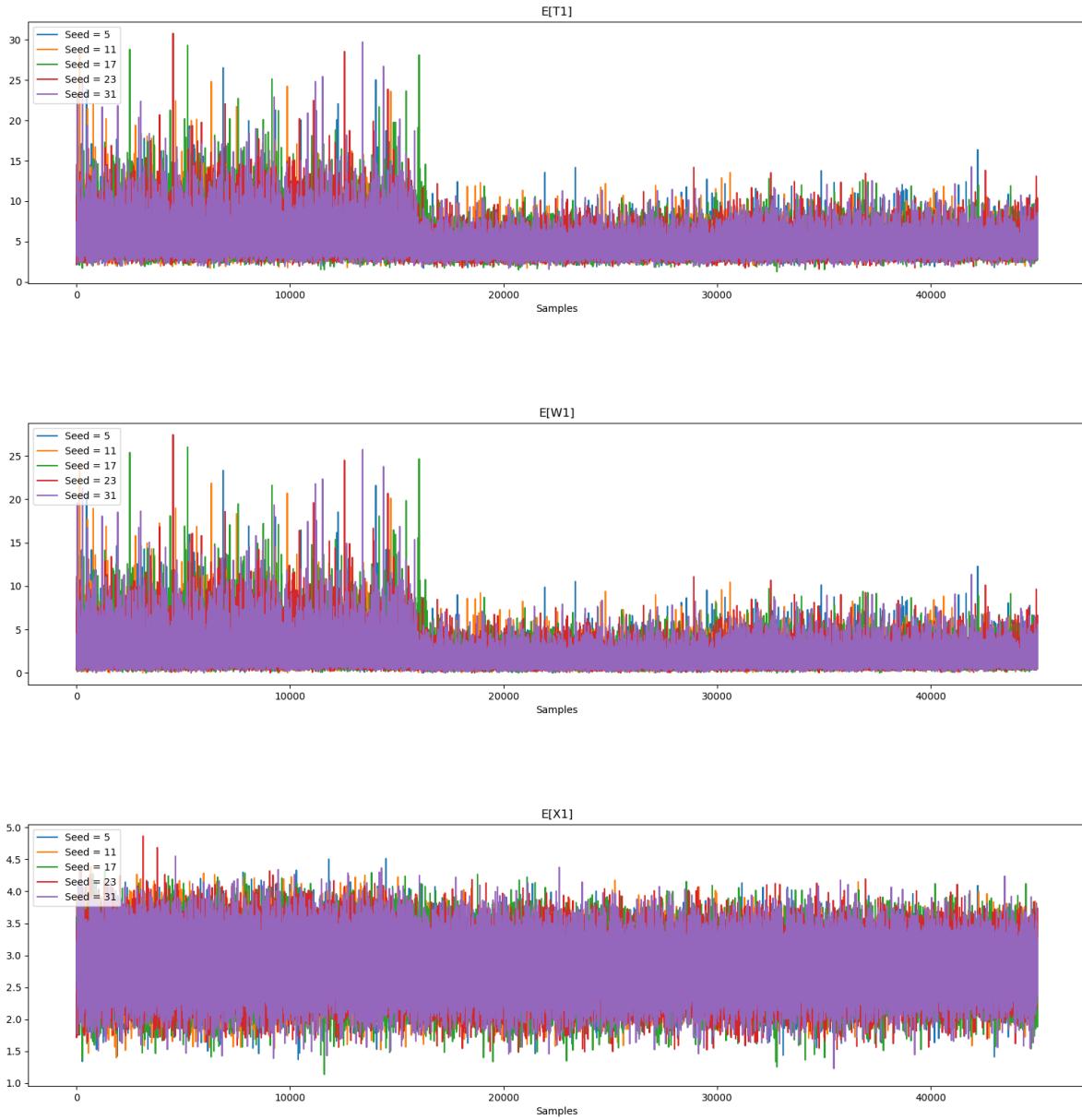


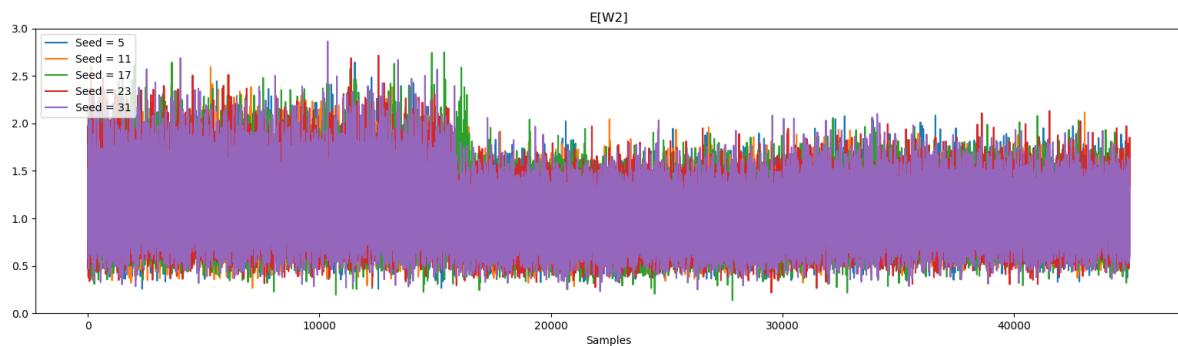
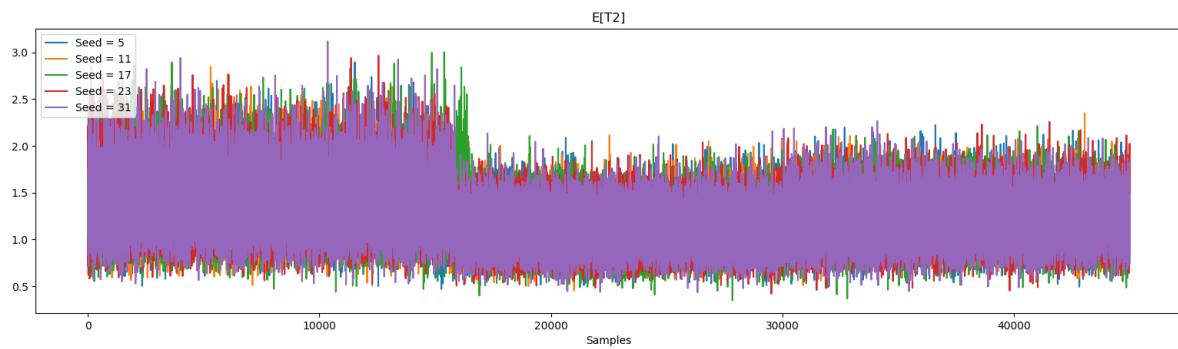
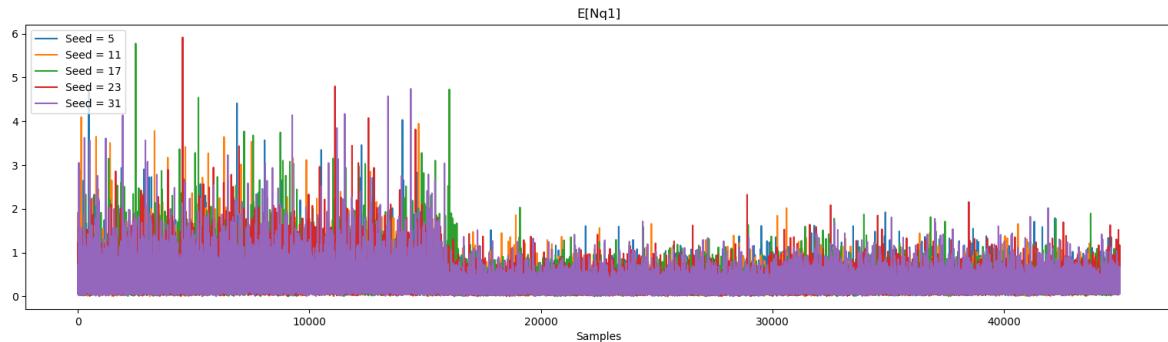


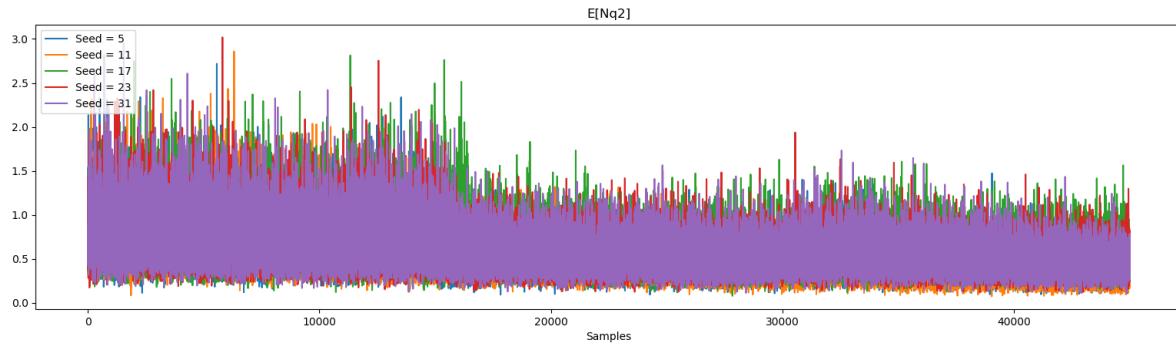
Podemos ver que os gráficos que melhor indicam uma convergência nos valores são os gráficos de $E[T1]$ e $E[T2]$. Esses gráficos indicam um equilíbrio a partir de, aproximadamente, 16000 rodadas, o que significam **3200000** amostras. Isso indica que, para alcançar um sistema em equilíbrio e otimizar o encontro de um intervalo de confiança pequeno, seria necessário ignorar um número muito grande de amostras. Por outro lado, qualquer número abaixo de 3200000 pode

ser usado como fase transitória a fim de diminuir o número total de amostras do simulador.

3.4 Para $\rho=0.4$ (não-preemptivo)

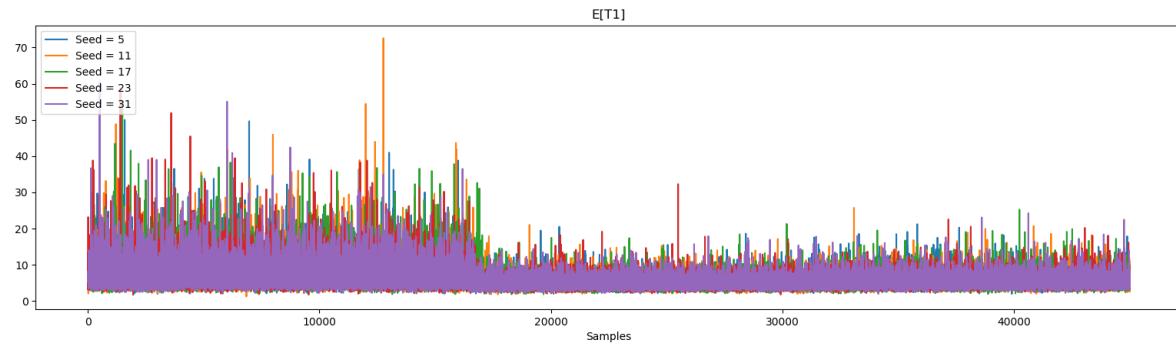


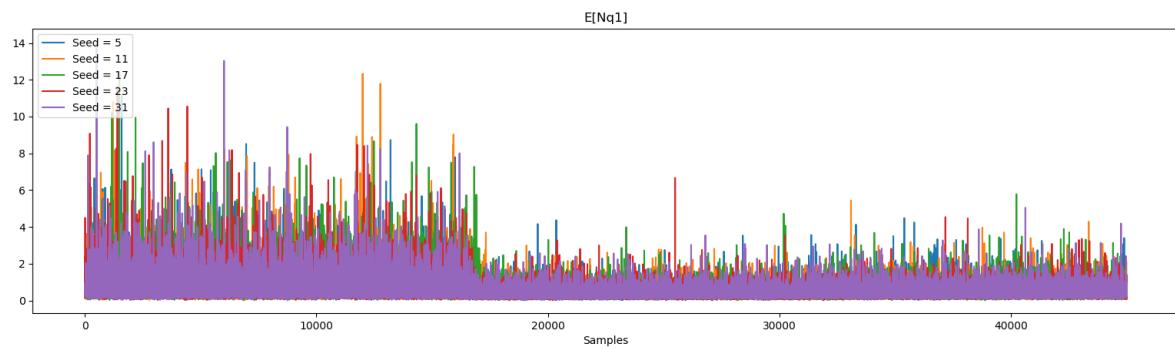
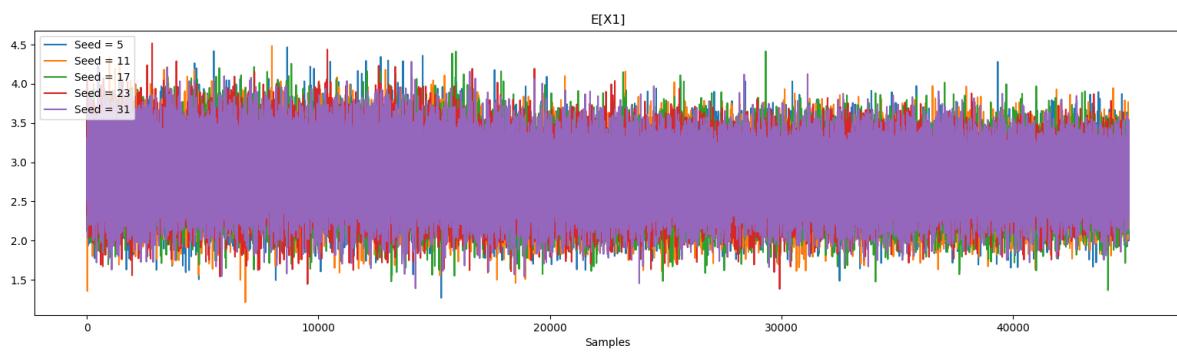
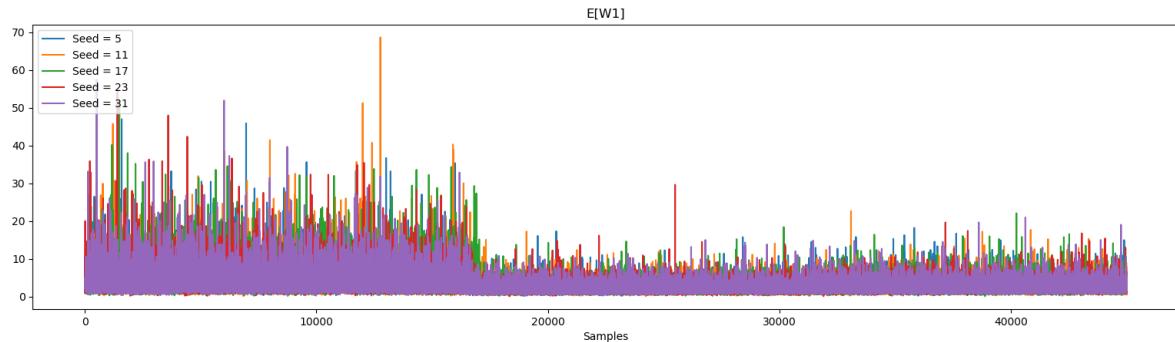


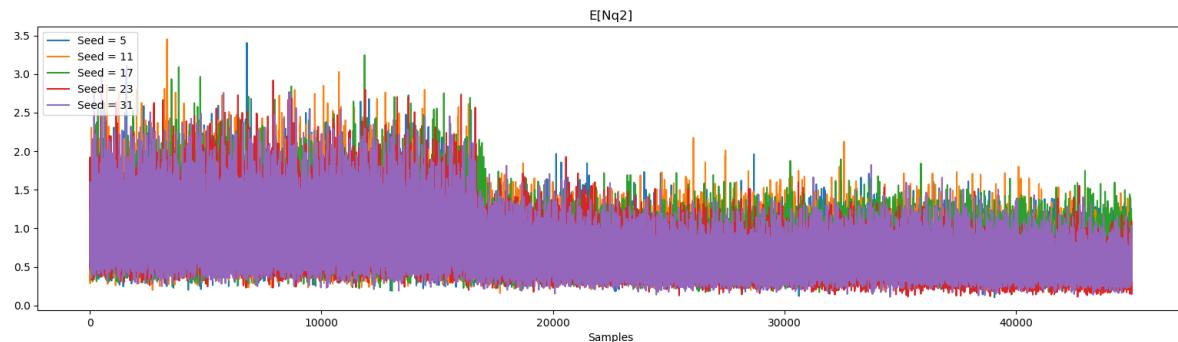
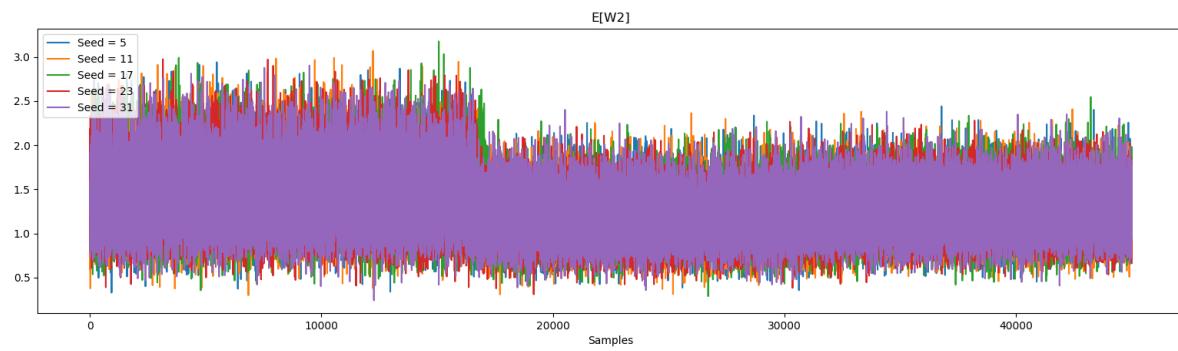
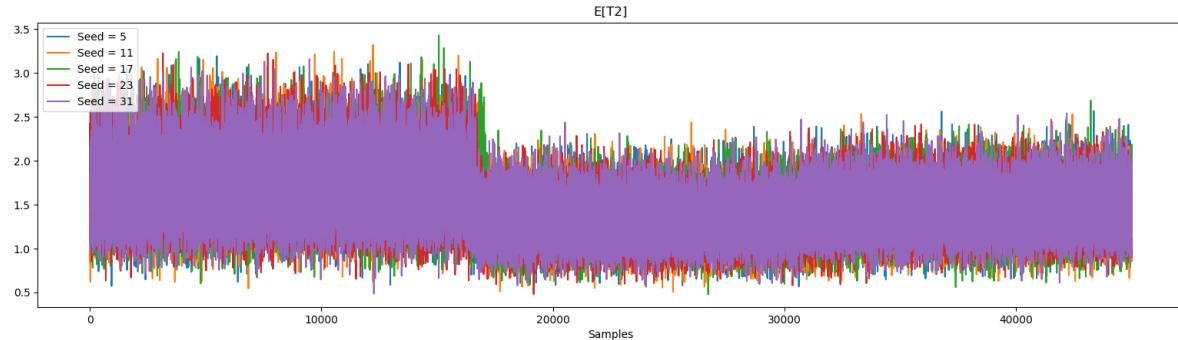


Podemos ver que os gráficos que melhor indicam uma convergência nos valores são os gráficos de $E[T1]$ e $E[T2]$. Esses gráficos indicam um equilíbrio a partir de, aproximadamente, 16000 rodadas, o que significam **3200000** amostras. Isso indica que, para alcançar um sistema em equilíbrio e otimizar o encontro de um intervalo de confiança pequeno, seria necessário ignorar um número muito grande de amostras. Por outro lado, qualquer número abaixo de 3200000 pode ser usado como fase transitória a fim de diminuir o número total de amostras do simulador.

3.5 Para $\rho=0.5$ (não-preemptivo)



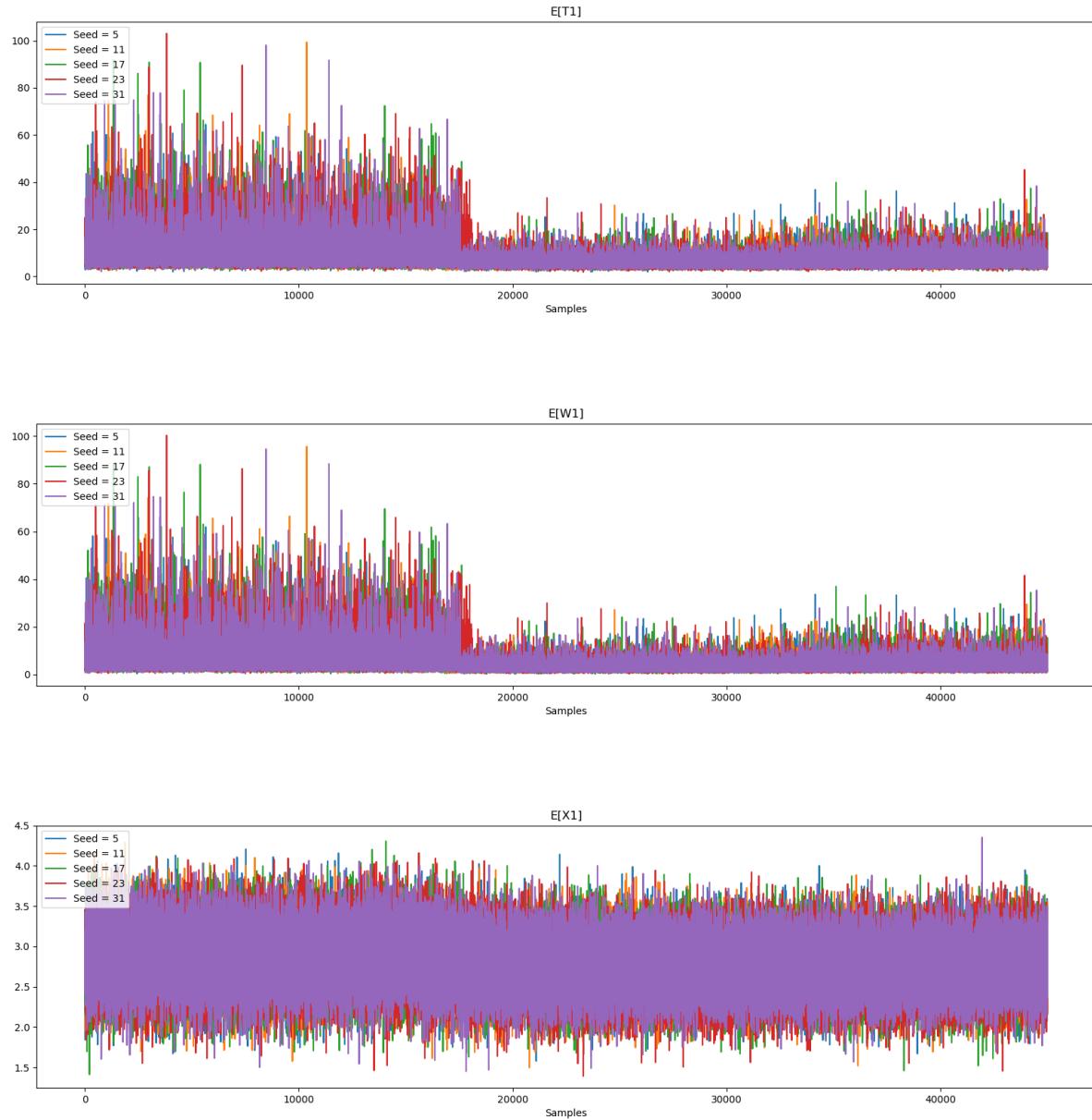


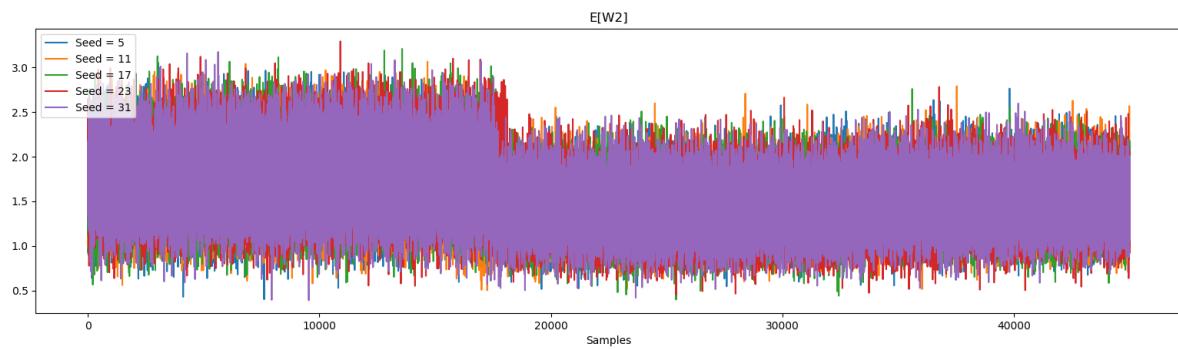
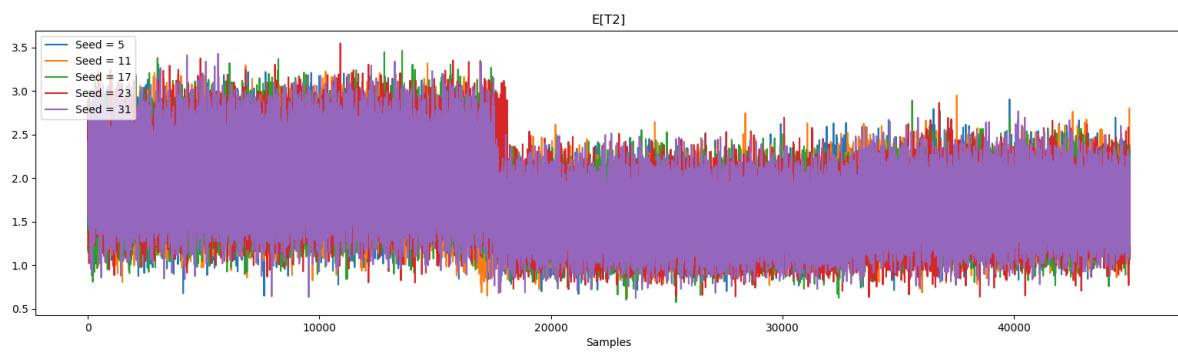
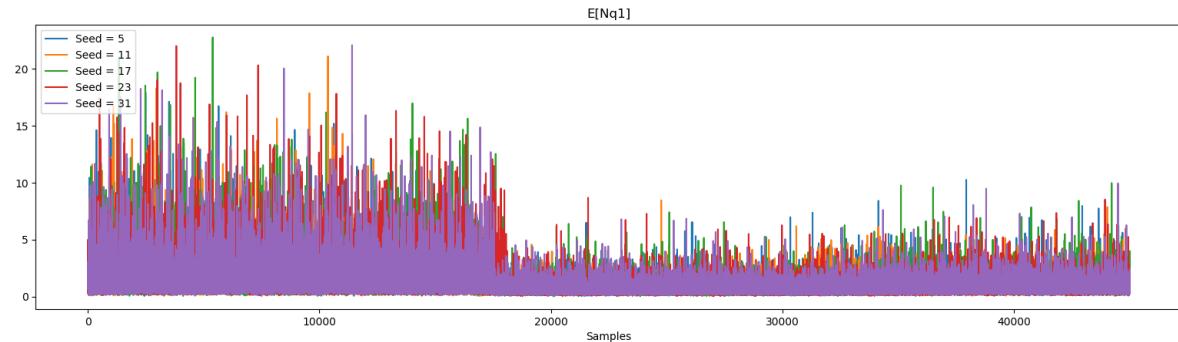


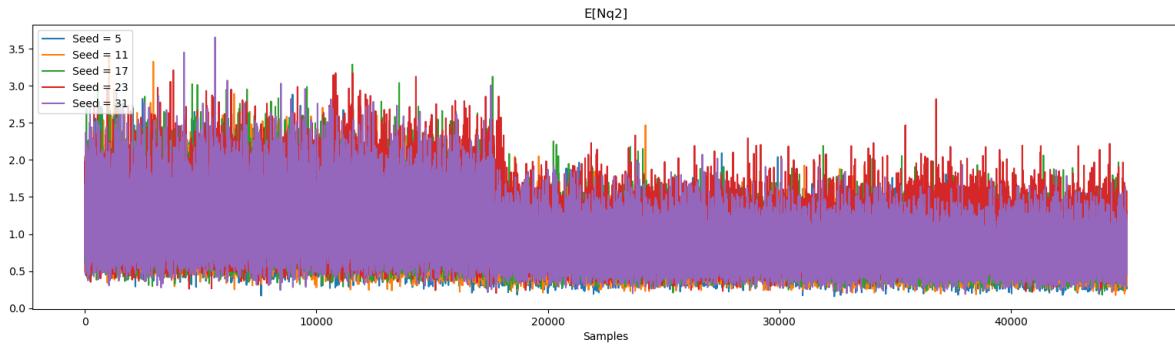
Podemos ver que os gráficos que melhor indicam uma convergência nos valores são os gráficos de $E[T_1]$ e $E[T_2]$. Esses gráficos indicam um equilíbrio a partir de, aproximadamente, 18000 rodadas, o que significam **3600000** amostras. Isso indica que, para alcançar um sistema em equilíbrio e otimizar o encontro de um intervalo de confiança pequeno, seria necessário ignorar um número muito grande de amostras. Por outro lado, qualquer número abaixo de 3600000 pode

ser usado como fase transitória ao fim de diminuir o número total de amostras do simulador.

3.6 Para $\rho=0.6$ (não-preemptivo)

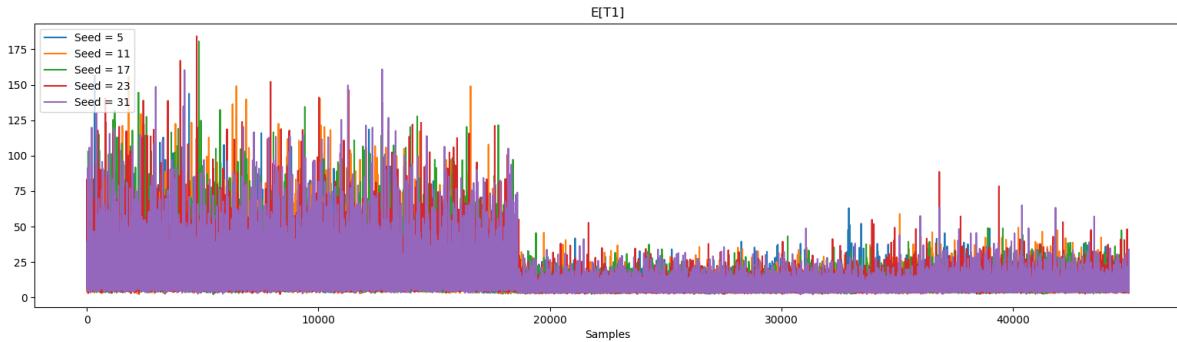


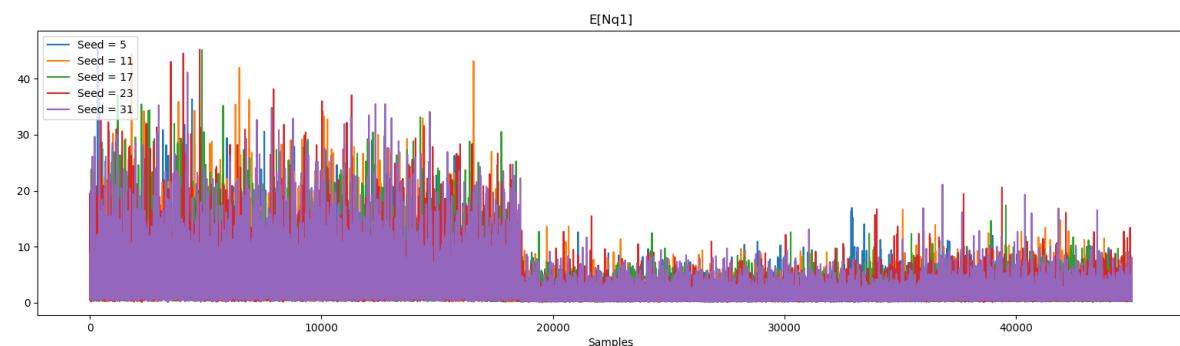
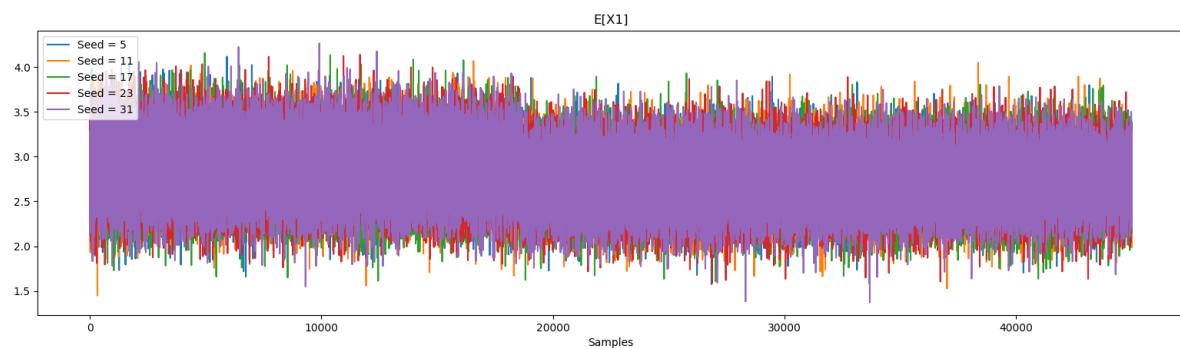
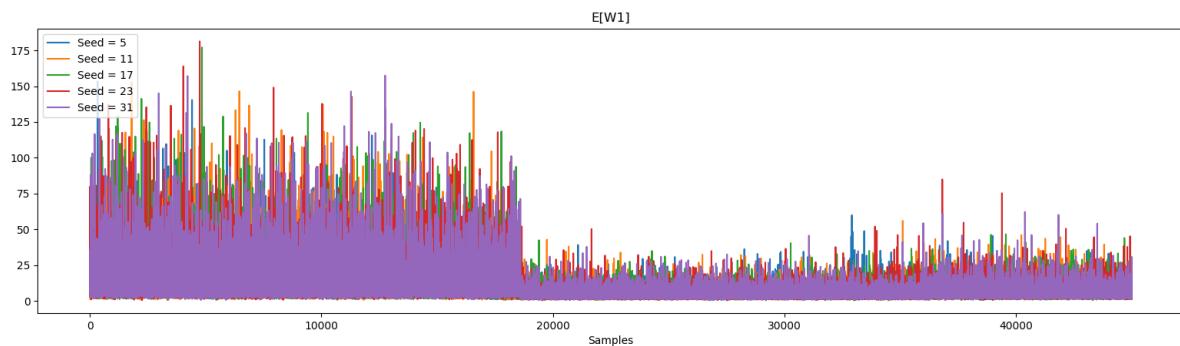


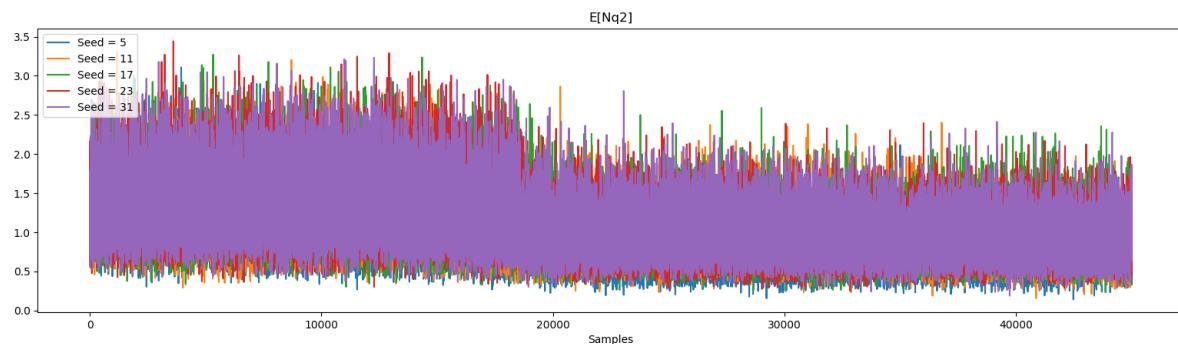
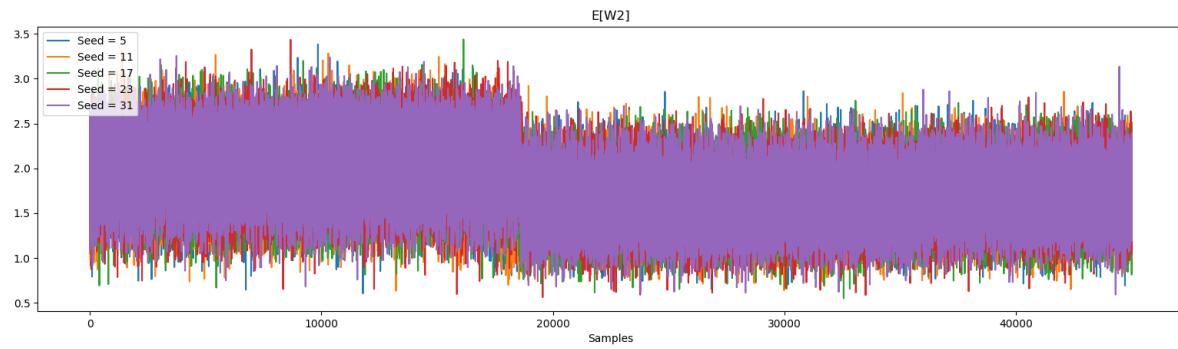
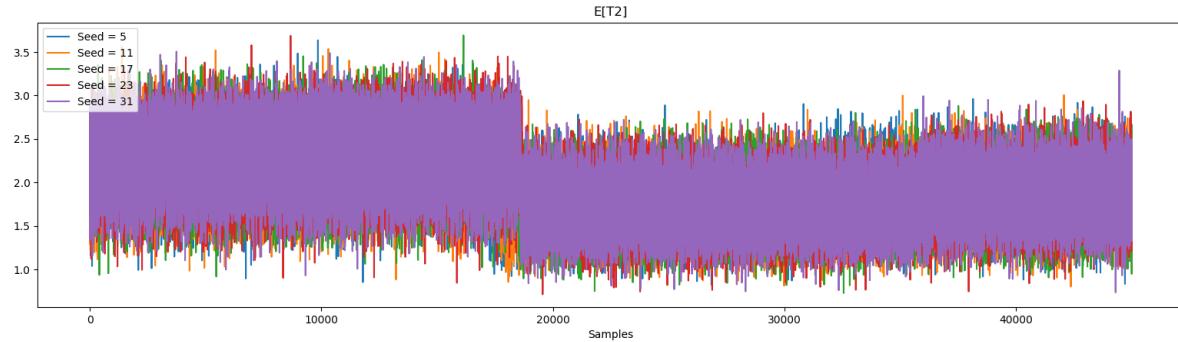


Podemos ver que os gráficos que melhor indicam uma convergência nos valores são os gráficos de $E[T1]$ e $E[T2]$. Esses gráficos indicam um equilíbrio a partir de, aproximadamente, 19000 rodadas, o que significam **3800000** amostras. Isso indica que, para alcançar um sistema em equilíbrio e otimizar o encontro de um intervalo de confiança pequeno, seria necessário ignorar um número muito grande de amostras. Por outro lado, qualquer número abaixo de 3800000 pode ser usado como fase transitória a fim de diminuir o número total de amostras do simulador.

3.7 Para $\rho=0.7$ (não-preemptivo)



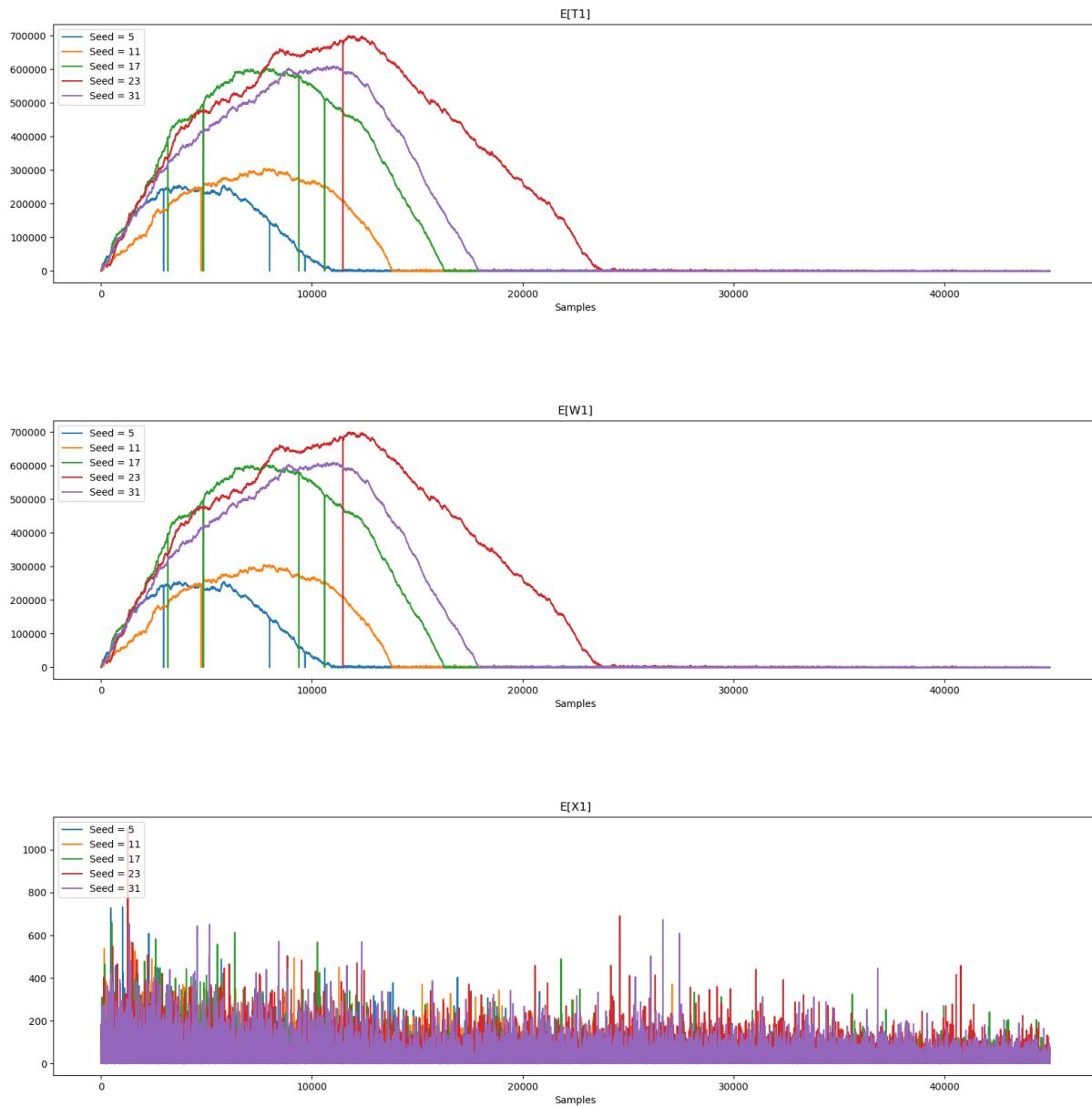


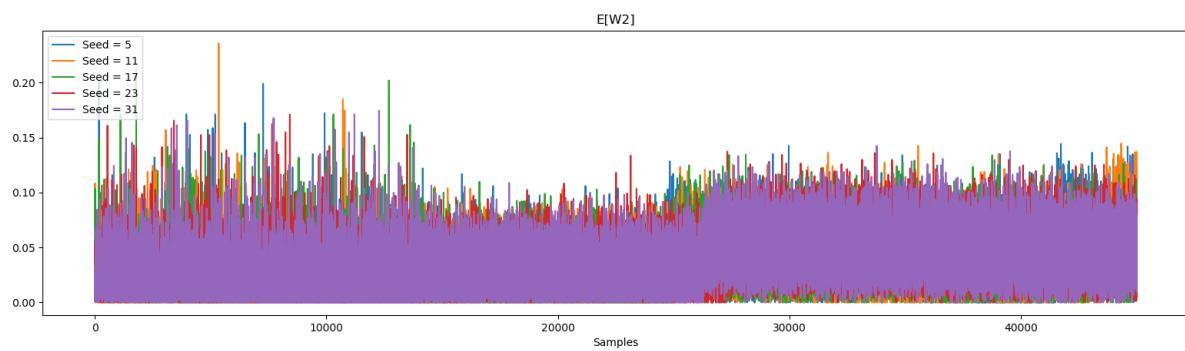
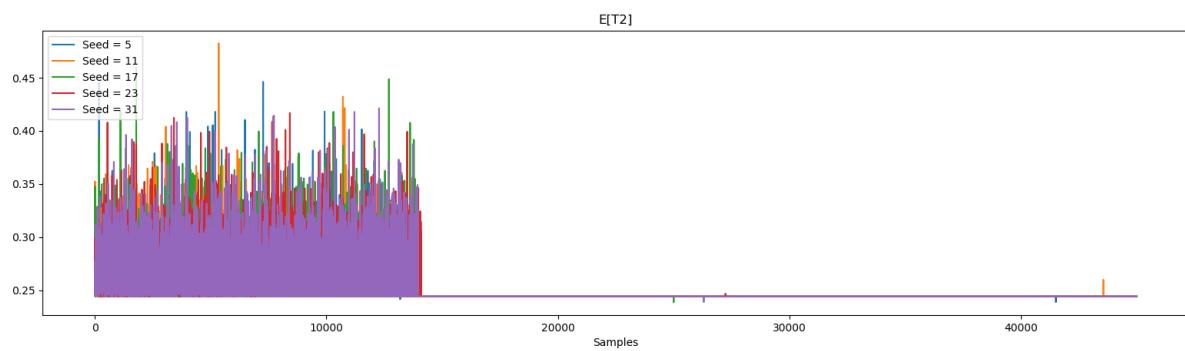
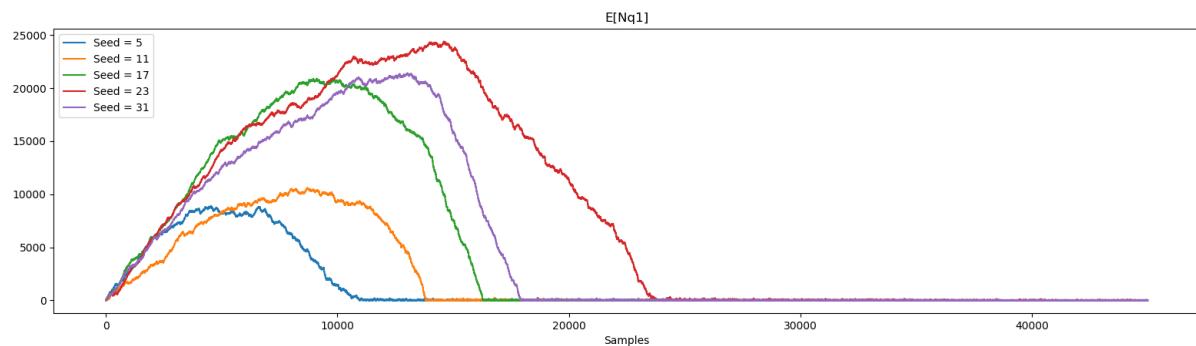


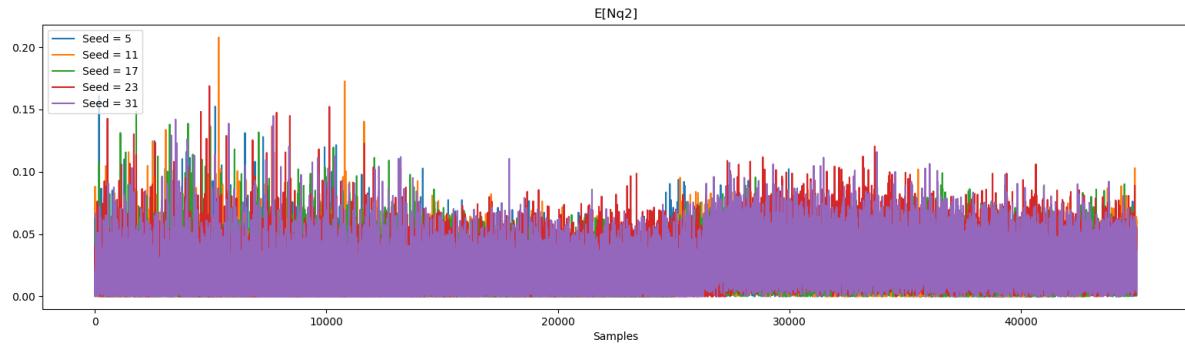
Podemos ver que os gráficos que melhor indicam uma convergência nos valores são os gráficos de $E[T_1]$ e $E[T_2]$. Esses gráficos indicam um equilíbrio a partir de, aproximadamente, 19000 rodadas, o que significam **3800000** amostras. Isso indica que, para alcançar um sistema em equilíbrio e otimizar o encontro de um intervalo de confiança pequeno, seria necessário ignorar um número muito grande de amostras. Por outro lado, qualquer número abaixo de 3800000 pode

ser usado como fase transitória a fim de diminuir o número total de amostras do simulador.

3.8 Para $\rho=0.1$ (preemptivo)

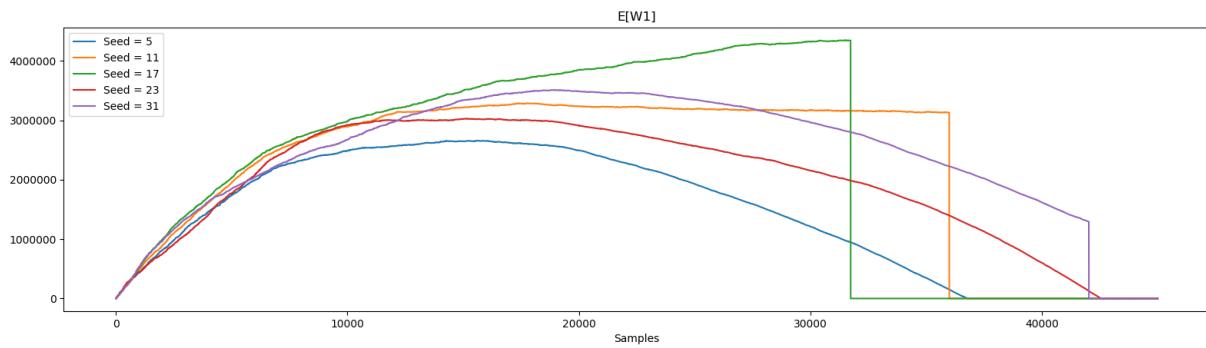
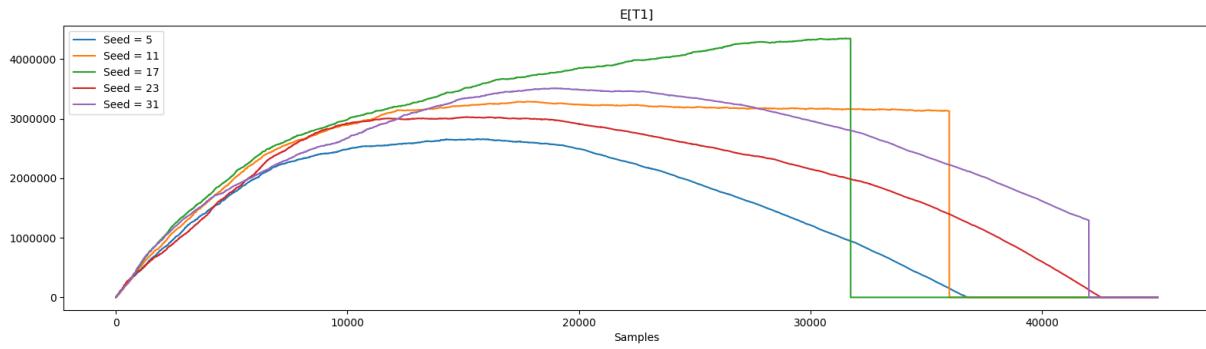


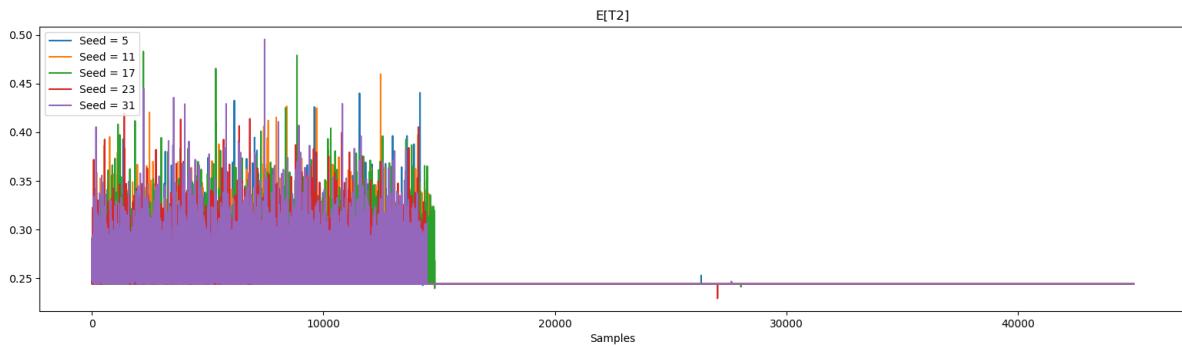
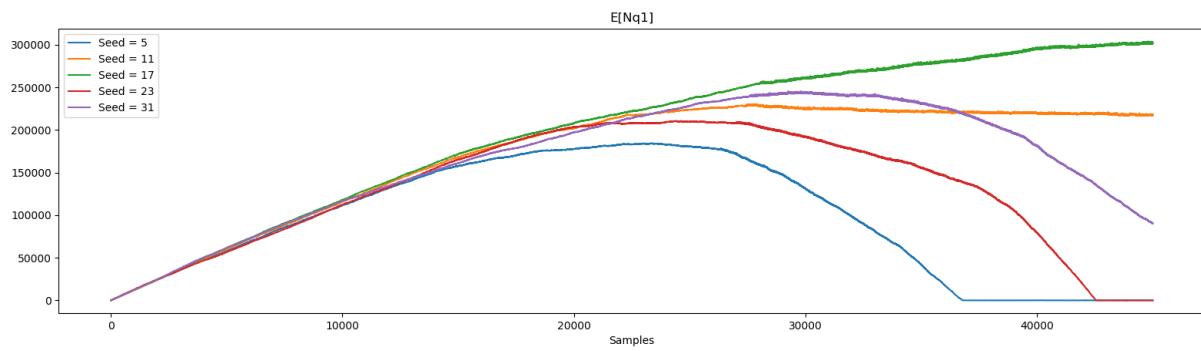
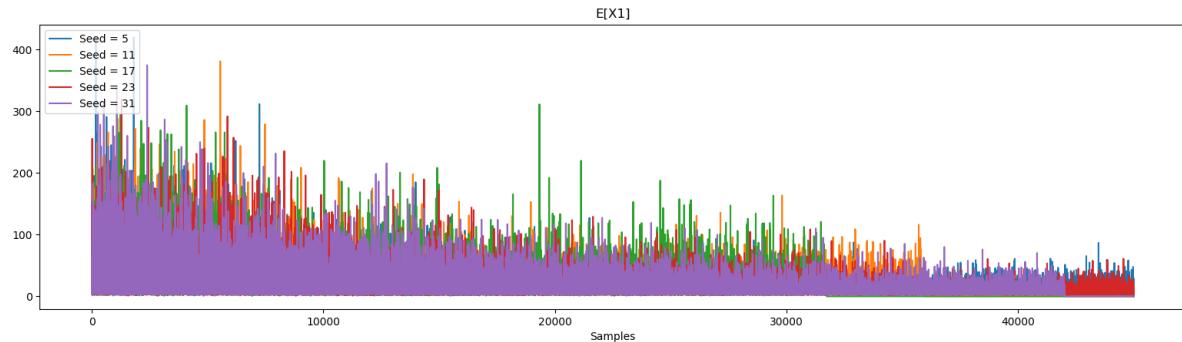


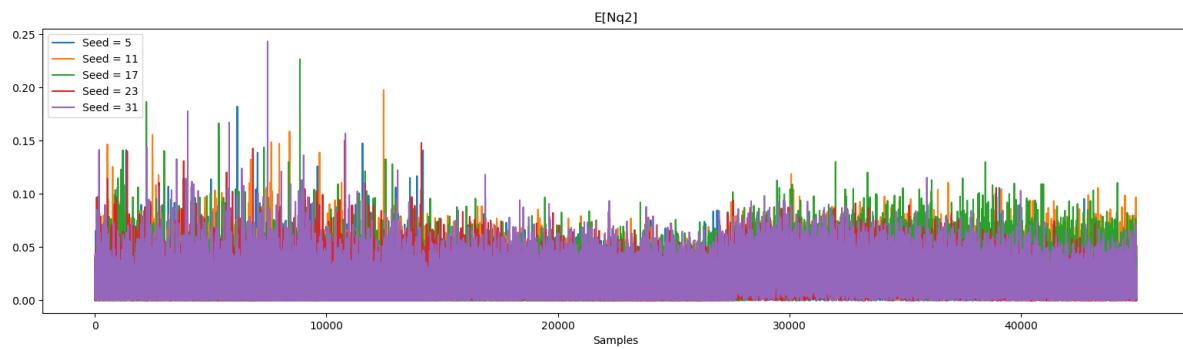
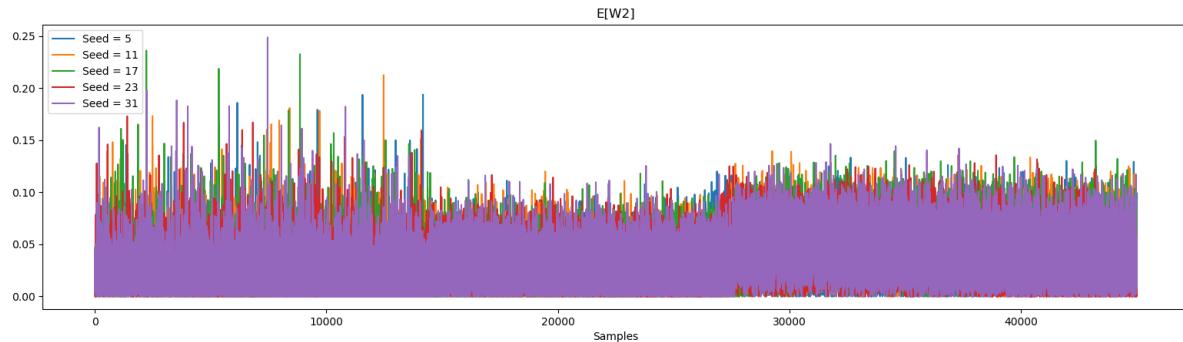


O gráfico $E[Nq1]$ é o mais claro. Podemos ver que o sistema entra em equilíbrio aproximadamente depois de 23000 rodadas, o que significam **4600000** amostras. Apesar de ser um número relativamente alto, a diminuição da variância é fundamental para diminuir o número total de amostras que o simulador precisará.

3.9 Para $\rho=0.2$ (preemptivo)

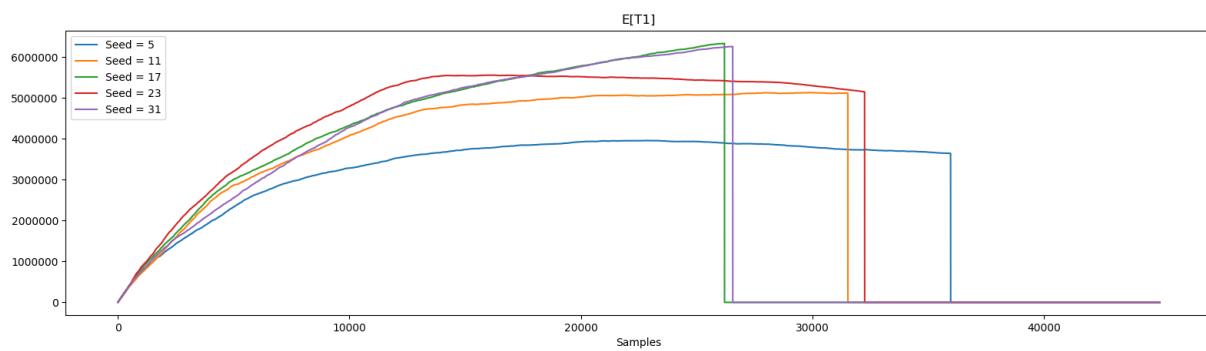


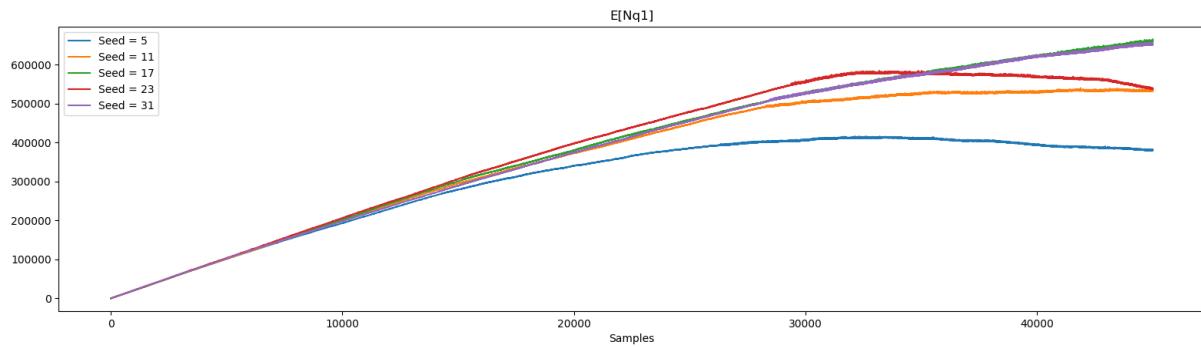
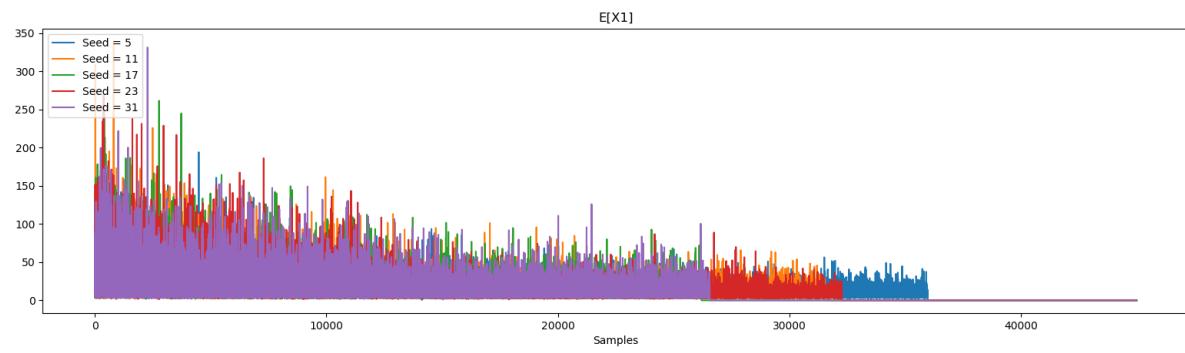
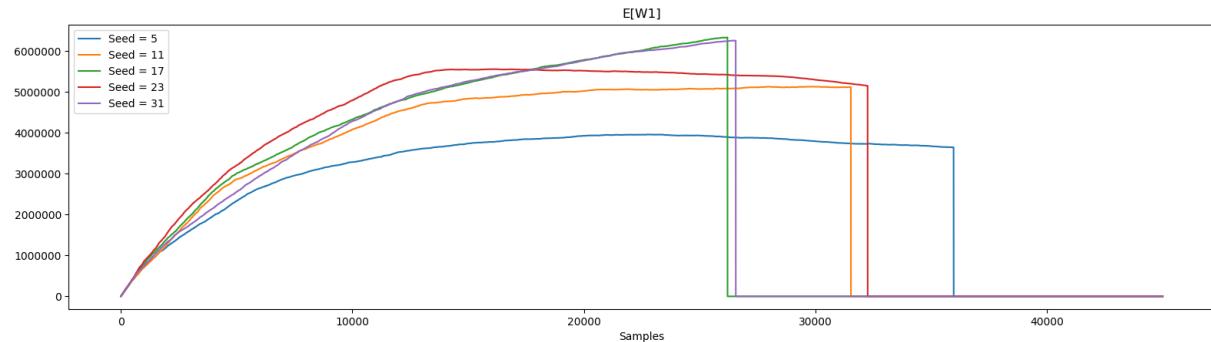


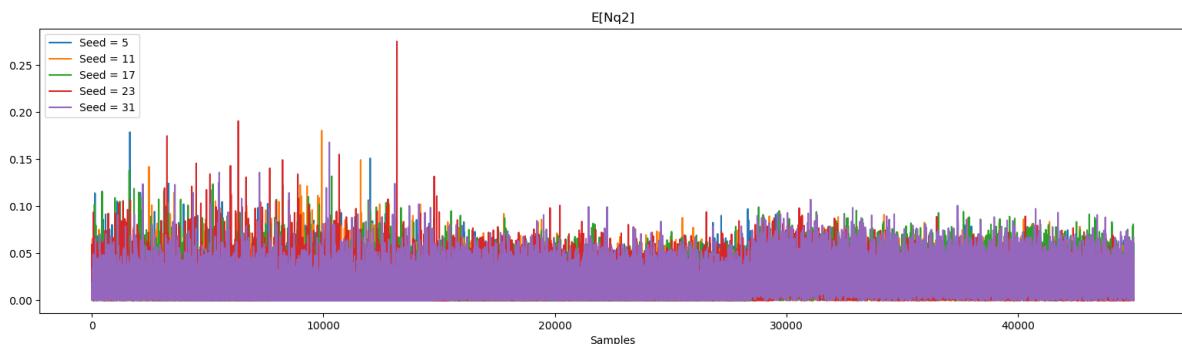
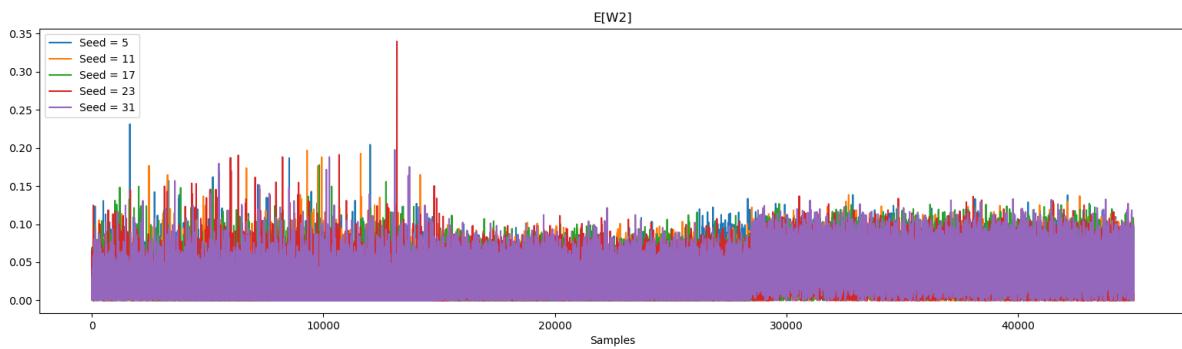
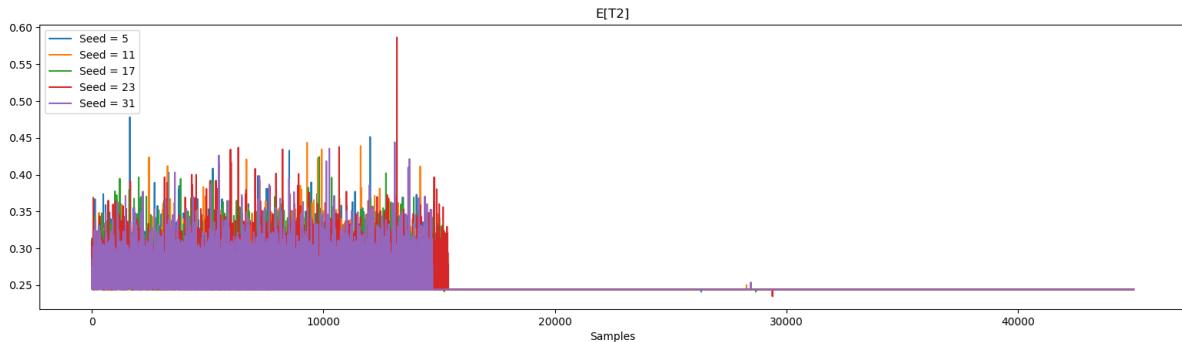


O gráfico $E[Nq1]$ é o mais claro. Podemos ver que o sistema começa a tender ao equilíbrio aproximadamente depois de 20000 rodadas, o que significam **4000000** amostras.

3.10 Para $\rho=0.3$ (preemptivo)

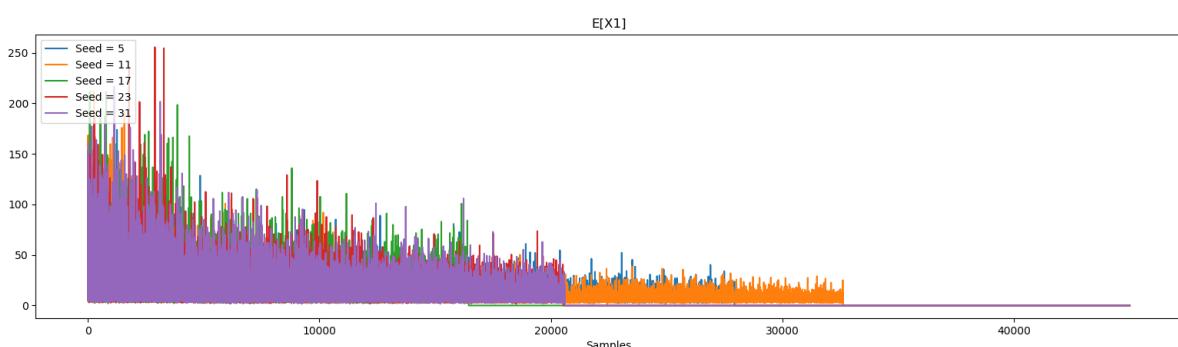
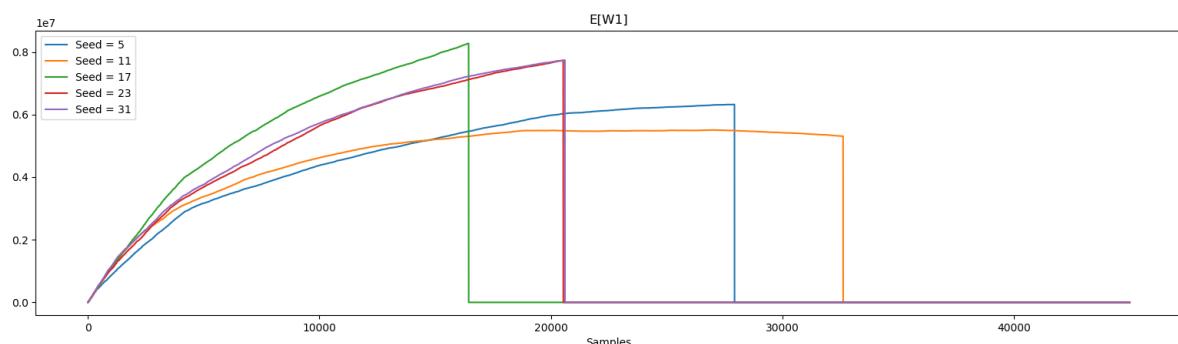
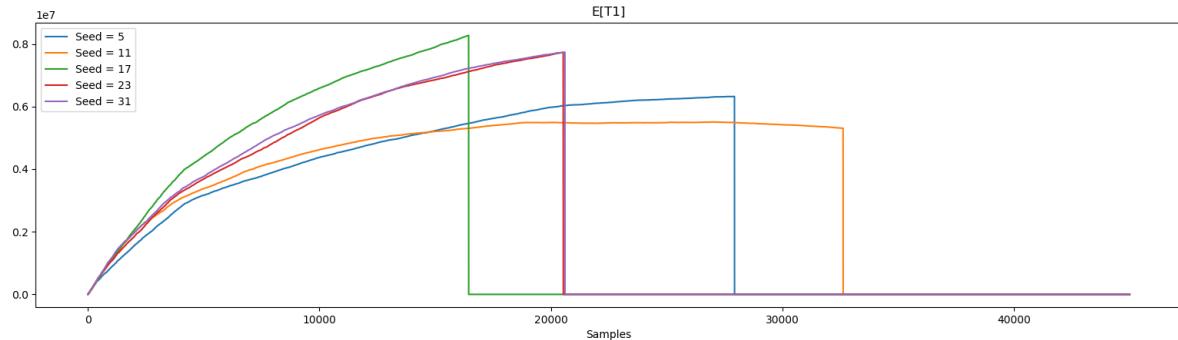


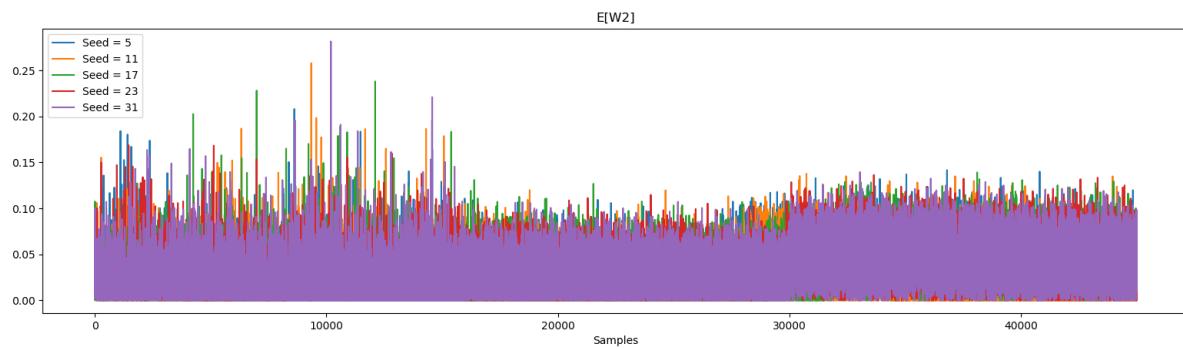
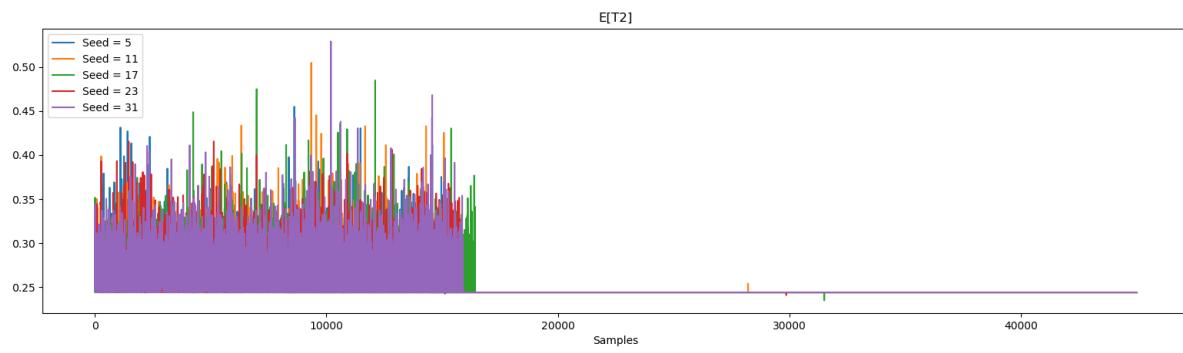
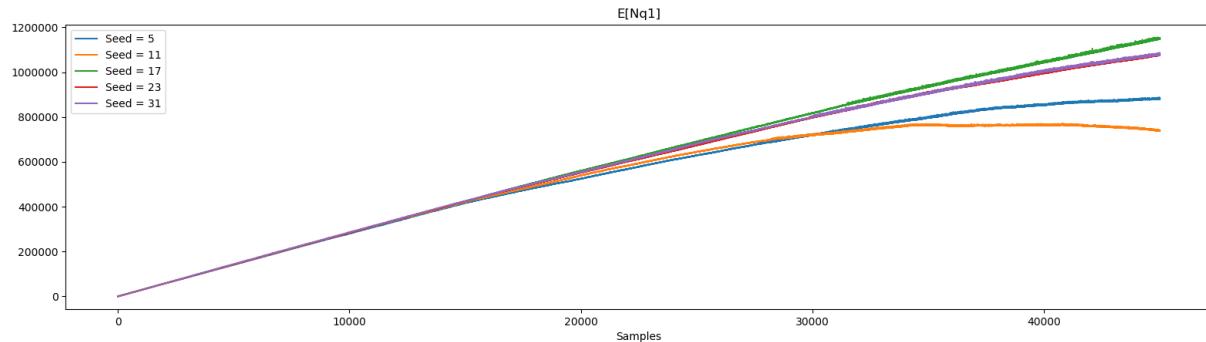


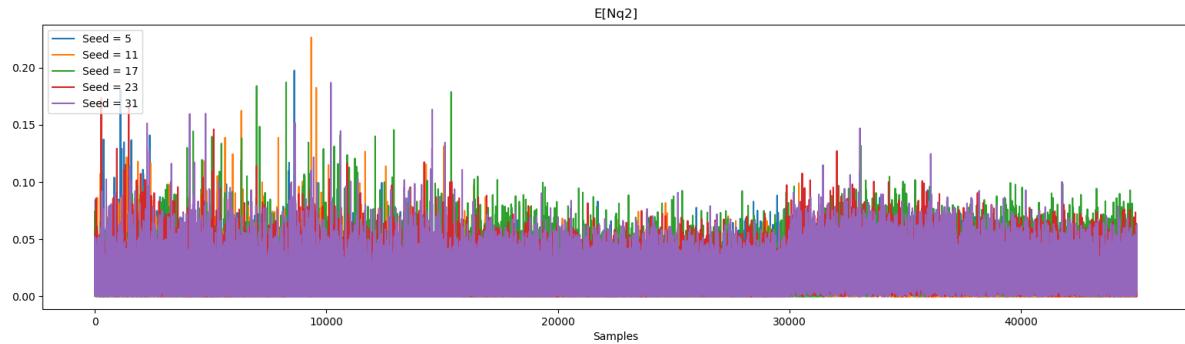


O gráfico $E[Nq_1]$ é o mais claro. Podemos ver que o sistema começa a tender ao equilíbrio aproximadamente depois de 30000 rodadas, o que significam **6000000** amostras.

3.11 Para $\rho=0.4$ (preemptivo)

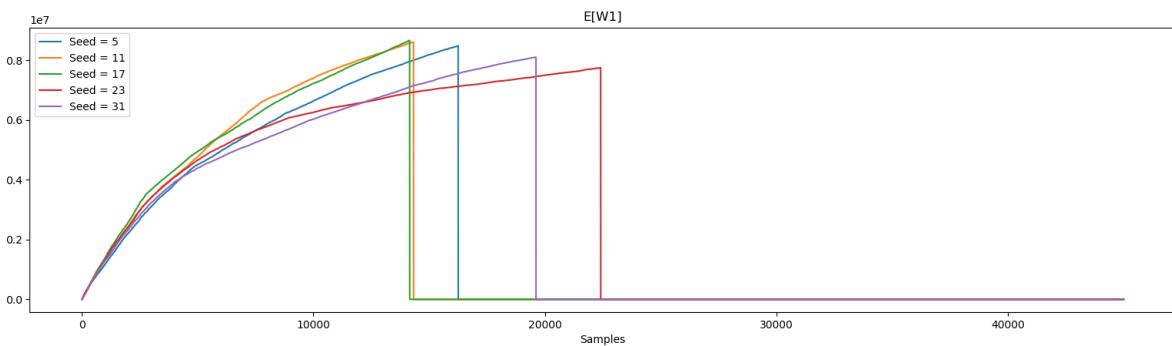
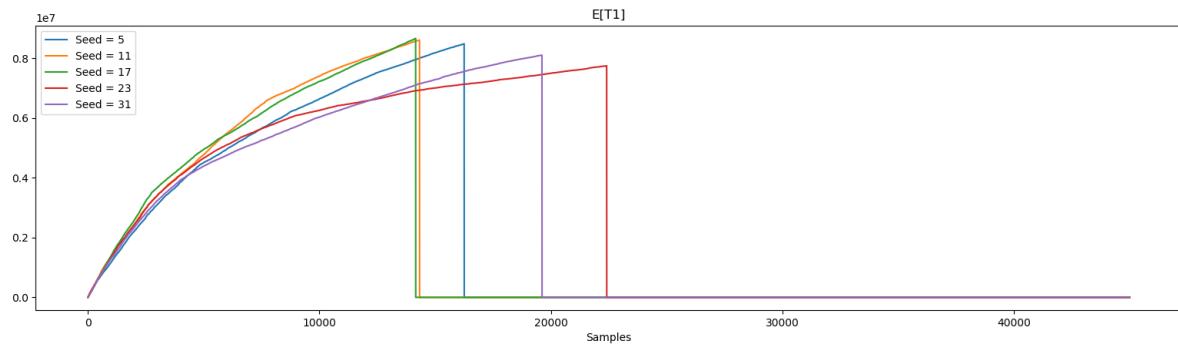


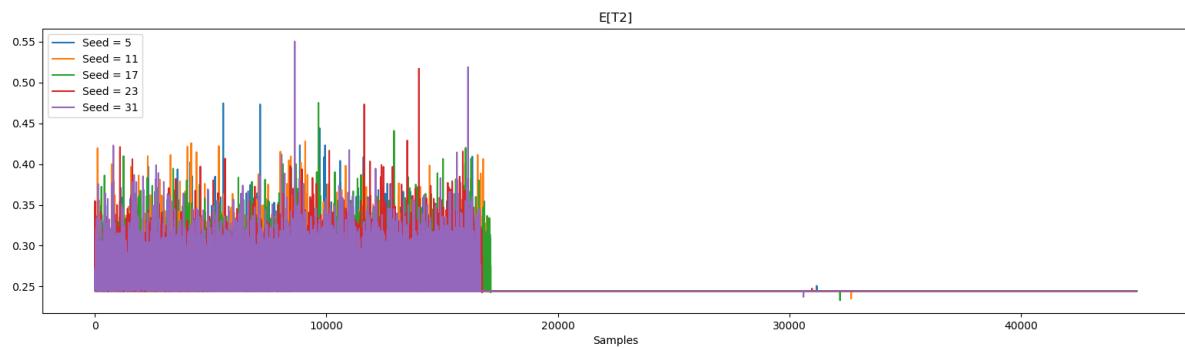
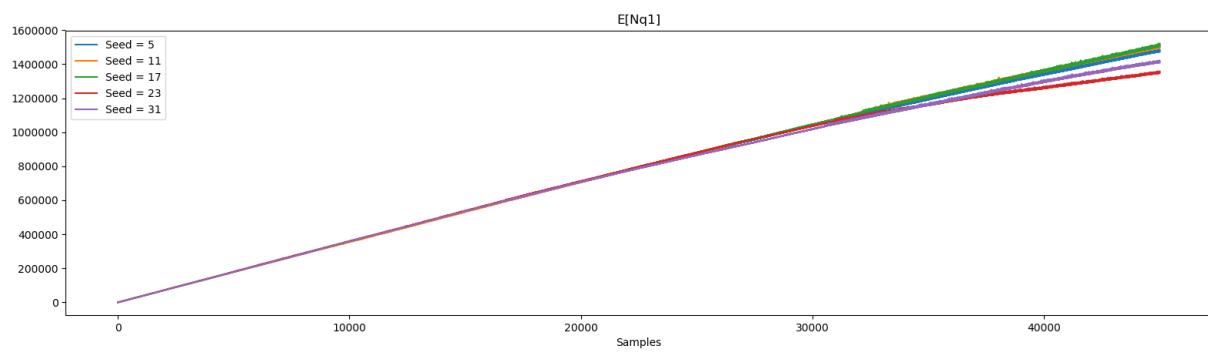
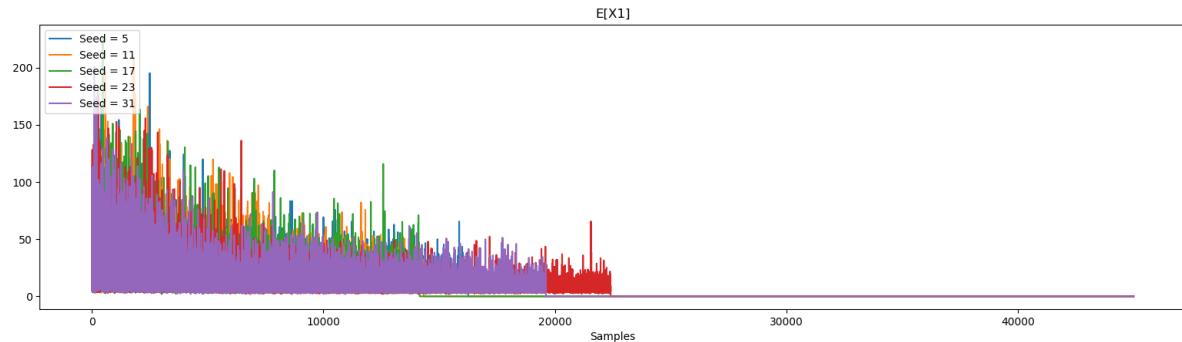


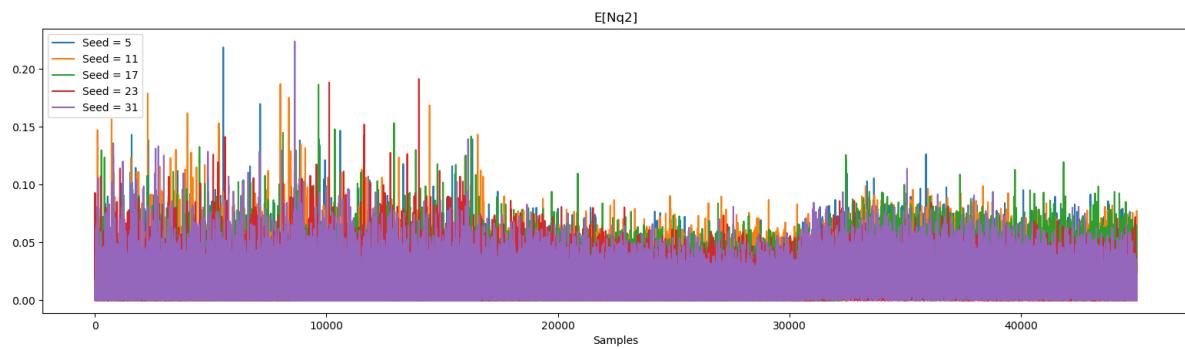
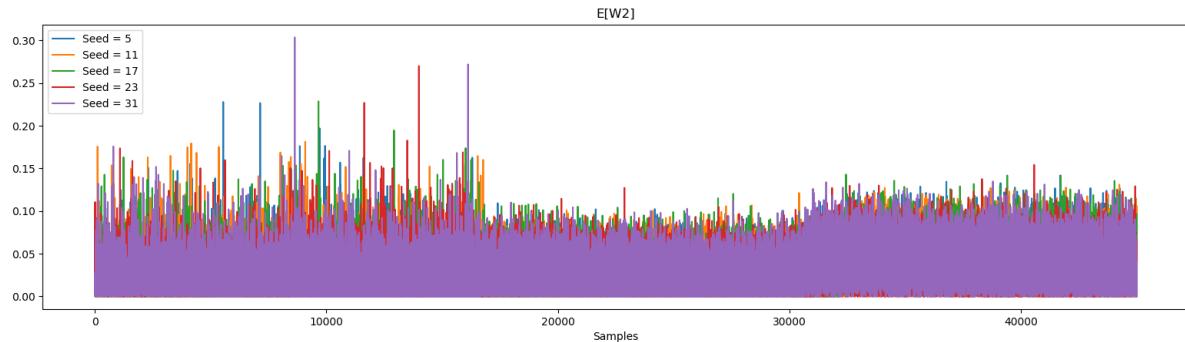


O gráfico $E[Nq1]$ é o mais claro. Podemos ver que o sistema começa a tender ao equilíbrio aproximadamente depois de 34000 rodadas, o que significam **6800000** amostras.

3.12 Para $\rho=0.5$ (preemptivo)

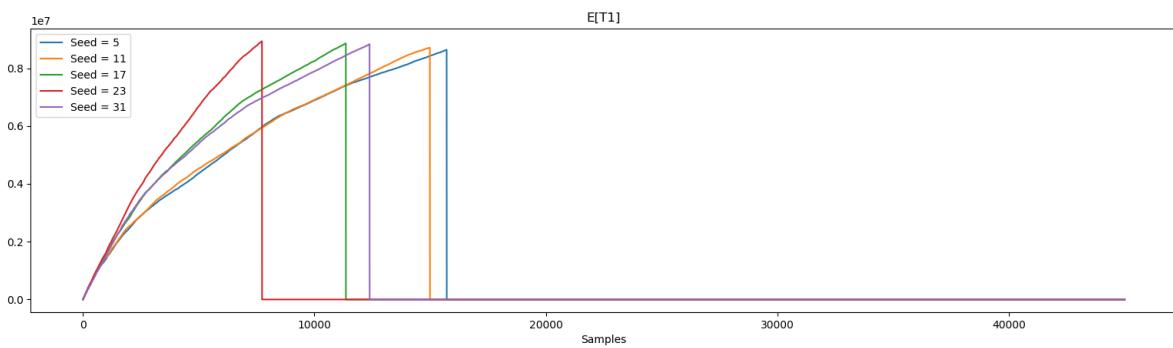


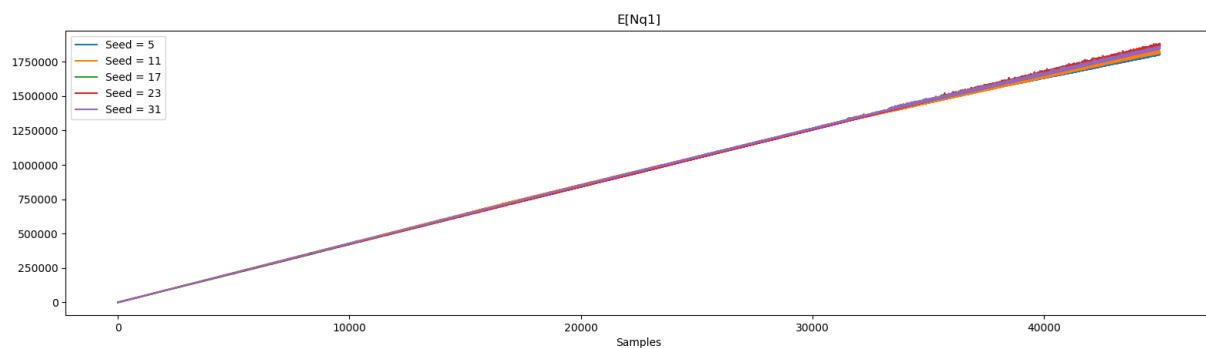
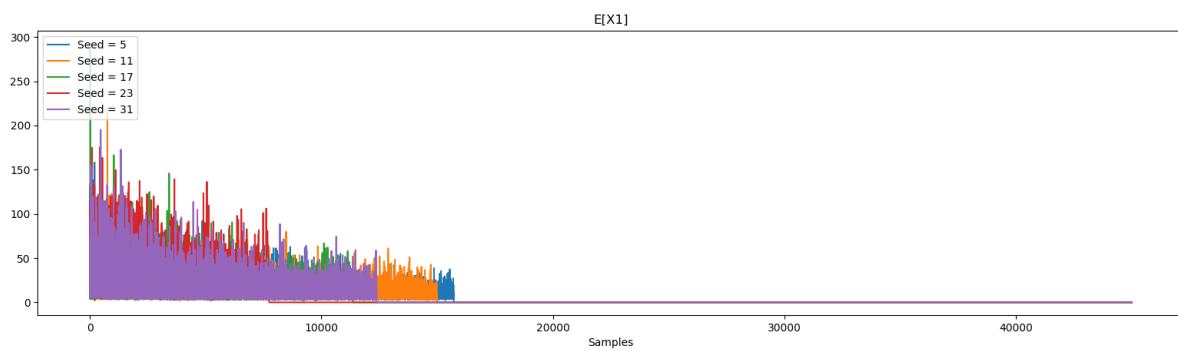
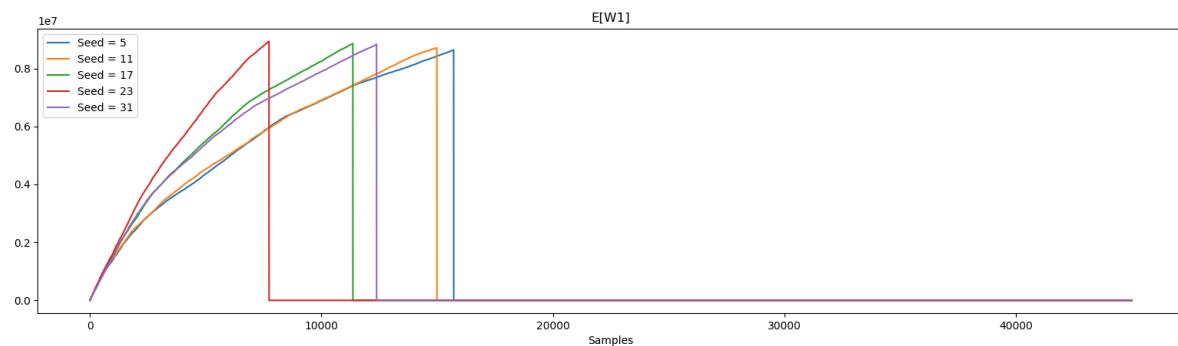


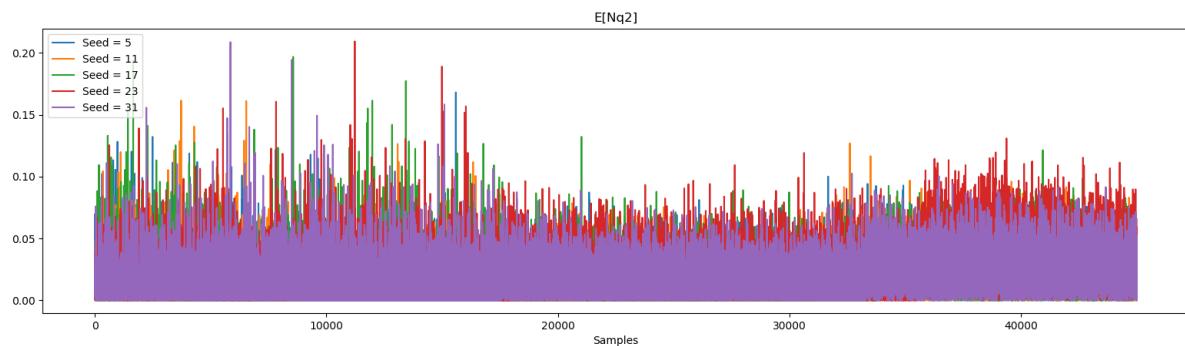
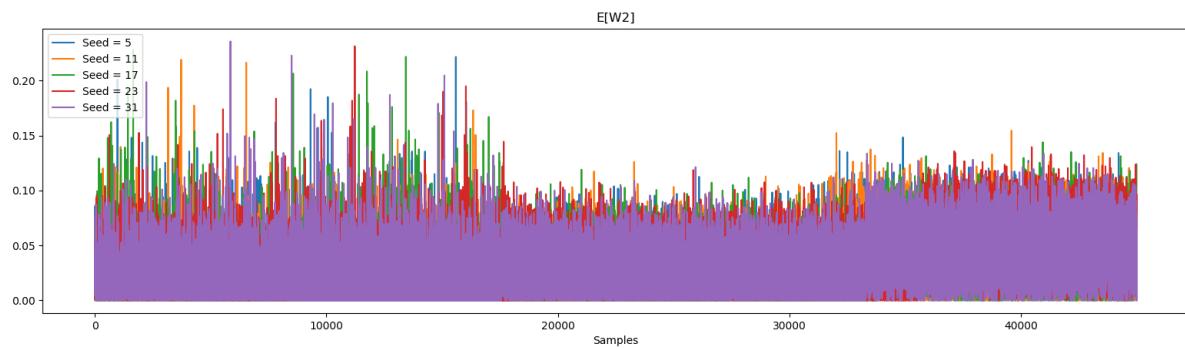
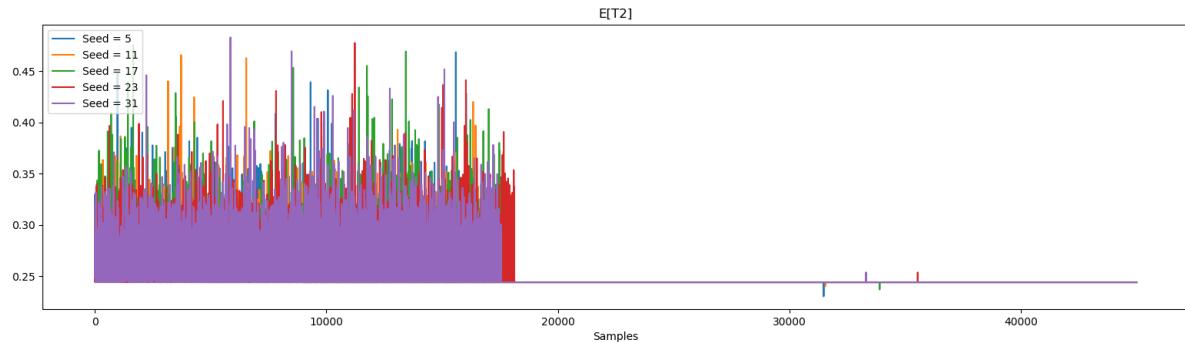


Mesmo plotando 9000000 amostras, não é possível prever quando o sistema tenderá ao equilíbrio.

3.13 Para $\rho=0.6$ (preemptivo)

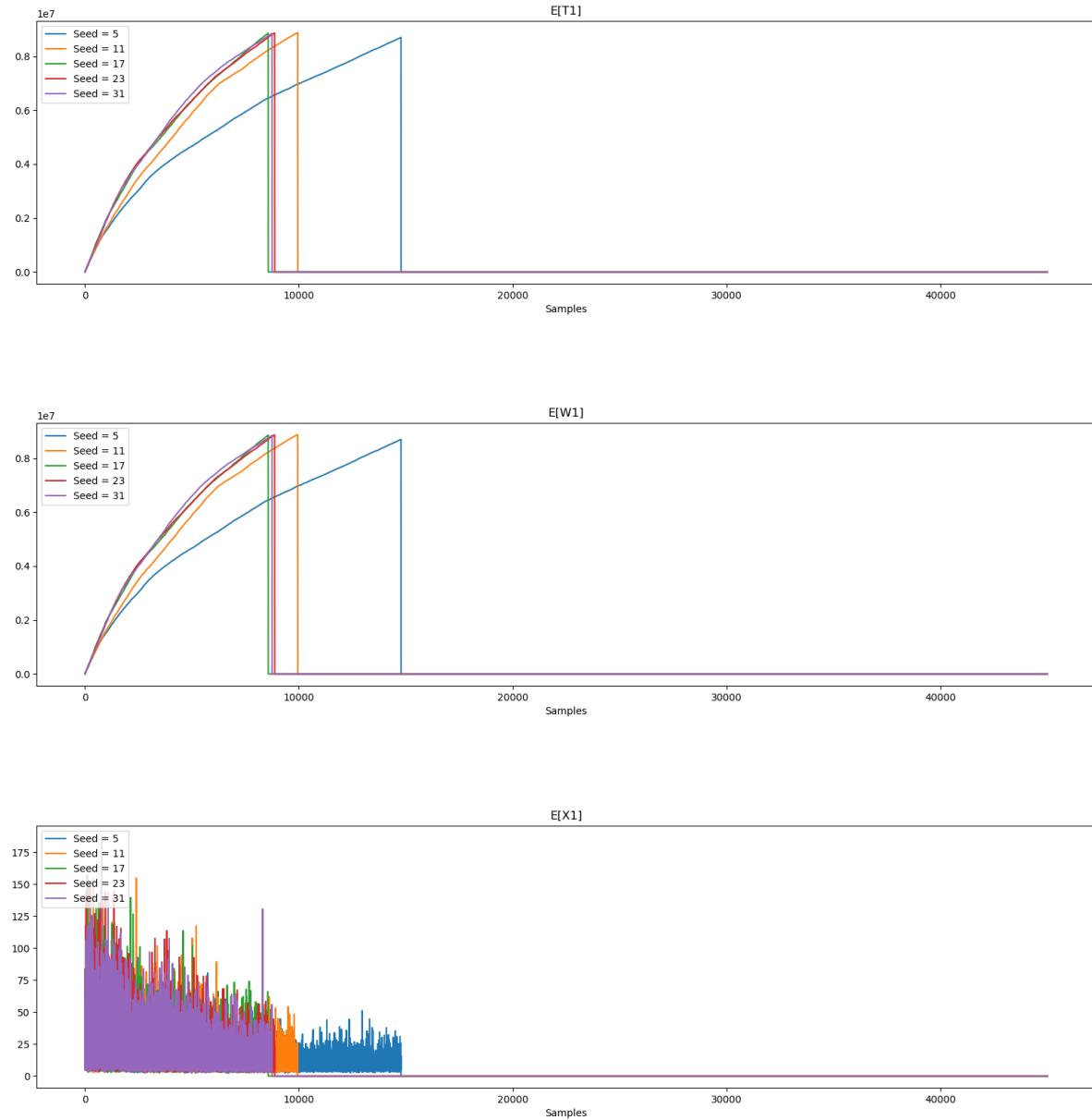


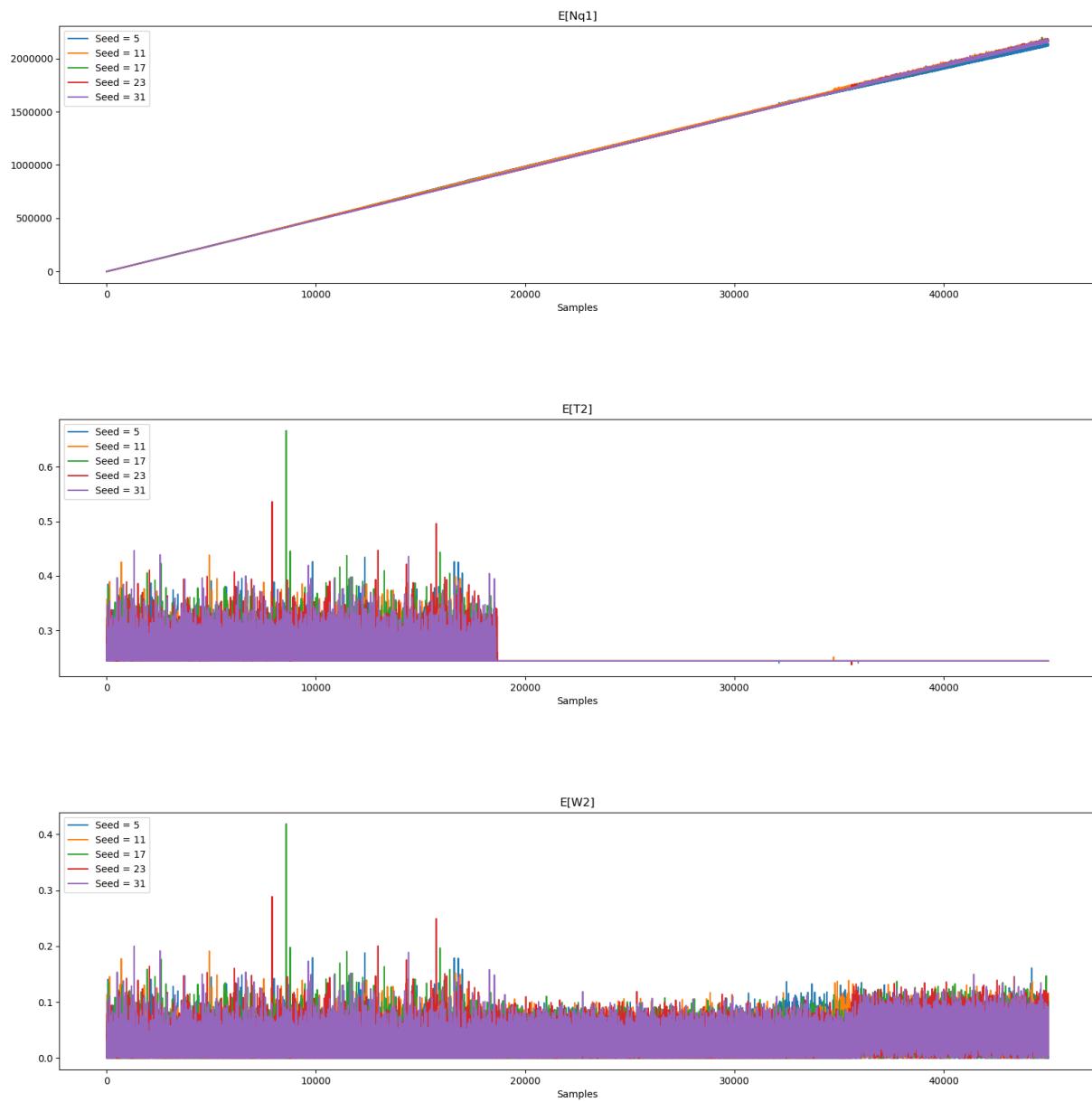


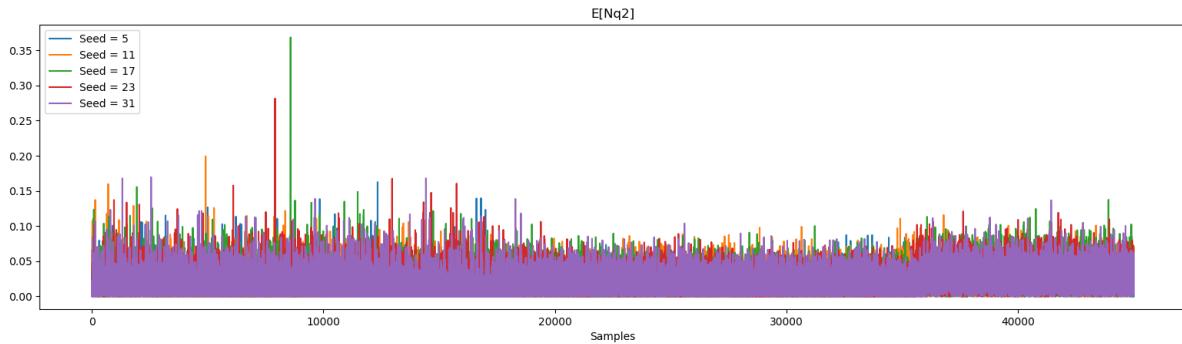


Mesmo plotando 9000000 amostras, não é possível prever quando o sistema tenderá ao equilíbrio.

3.14 Para $\rho=0.7$ (preemptivo)







Mesmo plotando 9000000 amostras, não é possível prever quando o sistema tenderá ao equilíbrio.

4 Listagem Documentada do Programa

4.1 simulator.cpp

```
#include "run_queues.cpp"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream>
using namespace std;

int main(){
    unsigned int seed;
    int transiente_period, customers_number, round_number;
    float utilization1;
    bool preemption, allow_logging;
    char input;
    // Gets the seed used for generation pseudo-random values
    cout << "Random seed: ";
    cin >> seed;
    // User chooses whether there will be preemption or not
    cout << "Preemption? (y/n): ";
    cin >> input;
    preemption = (input == 'y' || input == 'Y');
    // User chooses whether there will be data about the simulation stored in disk or not
    cout << "Allow logging? (y/n): ";
    cin >> input;
    allow_logging = (input == 'y' || input == 'Y');
    // It gets the transient period, the customers number, the rounds number and the
    utilization
}
```

```

    srand(seed);
    cout << "Transiente Period: ";
    cin >> transiente_period;
    cout << "Customers Number: ";
    cin >> customers_number;
    cout << "Round Number: ";
    cin >> round_number;
    cout << "Utilization 1: ";
    cin >> utilization1;

    // If the number is out of the limits, it doesn't execute the simulation
    if(transiente_period <= 0 || customers_number <= 0 || round_number <= 0
    || !(0.1 <= utilization1 && utilization1 <= 0.7))
        cout << "This isn't a valid value. Please, choose a value greater than 0 and
        between 0.1 and 0.7 for utilization.\n";
    else
        execution(transiente_period, customers_number, round_number, utilization1,
        preemption, allow_logging);

    return 0;
}

```

4.2 run queues.cpp

```

#include "Event.cpp"
#include <list>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream>
#include <time.h>
#include <fstream>
#include <vector>
#include <sstream>
using namespace std;

// Function that determines the number of voice packages (actual function)
int voice_package_number() {
    long double seed = (long double) rand1();
    if (seed == 1) seed = .999999; // seed cannot be 1 or there would be a log(0),
    which is infinite, so... we make it a little smaller
    return (int) ceil(log(1.0 - seed) / log(1 - 1.0L / 22.0));
}

// Creates the arrival event of a data customer
Event create_data(int customer_id, float simulation_time, float lambda){

```

```

        double arrival_time = exponential(lambda);
        Customer customer = Customer(customer_id, DATA, simulation_time + arrival_time);
        Event event = Event(simulation_time + arrival_time, customer, ARRIVAL);
        return event;
    }

    // Creates the arrival event of a voice customer
    Event create_voice(int customer_id, float simulation_time, float offset, int channel_id){
        Customer customer = Customer(customer_id, VOICE, simulation_time + offset, channel_id);
        Event event = Event(simulation_time + offset, customer, ARRIVAL, channel_id);
        return event;
    }

    // Creates the silence period event
    Event create_silence_period(float simulation_time, float offset, int channel_id){
        double arrival_time = exponential(1.0/650);
        Event event = Event(simulation_time + arrival_time + offset, channel_id);
        return event;
    }

    // Creates the exit event of any package
    Event remove_package(float simulation_time, Customer customer){
        Event event = Event(simulation_time + customer.service_time, customer, EXIT);
        return event;
    }

    // It runs the rounds of the simulation
    void rounds(unsigned int seed, int transient_period, int customers_number,
    int round_number, float lambda, bool preemption, bool allow_logging){
        queue* data_traffic = queue_create(); // Queue where the data packages are stored
        queue* voice_traffic = queue_create(); // Queue where the voice packages are stored

        double simulation_time = 0; // Current time in the simulator
        double last_time[30]; //Keeps the last time a package of each specific channel entered
        the server

        double total_time = 0;

        for (int i = 0; i < 30; i++){
            last_time[i] = 0;
        }

        // Gets the intervals of each channel;
        vector <double> intervals;

        // Since we must ignore the first transient_period customers, only customers considered

```

```

will be the ones with id >= 0
// and we'll start counting from -transient_period
Customer::total_customers = -transient_period;

Customer customer_being_served = Customer(); // The customer currently in the server.
NONE type = no customer there.

    // Expectations/Averages
float T1[round_number+1]; // Average time a data package stays in the system
float W1[round_number+1]; // Average time a data package stays in the queue
float X1[round_number+1]; // Average time a data package stays in the server
    float Nq1[round_number+1]; // Average number of data packages in the queue
float T2[round_number+1]; // Average time a voice package stays in the system
float W2[round_number+1]; // Average time a voice package stays in the queue
    float Nq2[round_number+1]; // Average number of voice packages in the queue

    // Interval between packages Estimator
    float EDelta[round_number+1];
    float VDelta[round_number+1];

    for (int i = 0; i < round_number+1; i++) {
T1[i] = 0;
W1[i] = 0;
X1[i] = 0;
Nq1[i] = 0;
T2[i] = 0;
W2[i] = 0;
Nq2[i] = 0;

EDelta[i] = 0;
VDelta[i] = 0;
}

    // Averages, lower limits and upper limits of the confidence intervals
float ET1 = 0, lower_T1, upper_T1;
float EW1 = 0, lower_W1, upper_W1;
float EX1 = 0, lower_X1, upper_X1;
float ENq1 = 0, lower_Nq1, upper_Nq1;
float ET2 = 0, lower_T2, upper_T2;
float EW2 = 0, lower_W2, upper_W2;
float ENq2 = 0, lower_Nq2, upper_Nq2;
float EEDelta = 0, lower_EDelta, upper_EDelta;
float EVDelta = 0, lower_VDelta, upper_VDelta;

    // Standard deviations of the confidence intervals
float ST1 = 0;

```

```

float SW1 = 0;
float SX1 = 0;
float SNq1 = 0;
float ST2 = 0;
float SW2 = 0;
float SNq2 = 0;
float SEDelta = 0;
float SVDelta = 0;

// Variables used for the areas method (Data Queue)
double time_data = 0; // data queue timestamps
int size_data; // data queue sizes

// Variables used for the areas method (Voice Queue)
double time_voice = 0; // voice queue timestamps
int size_voice; // voice queue sizes

// People that came out of the system coming from both Queues;
int out1 = 0; // data packages
int out2 = 0; // voice packages
int out = 0; // out1 + out2

// How many voice packages each channel needs to send before entering silence period.
int voice_channels[30];
for (int i = 0; i < 30; i++) voice_channels[i] = 0;

list<Event> event_list;

// Inserts the event of the first data package arrival
list_insert(event_list, create_data(-99999, simulation_time, lambda));

// VOICE CHANNELS
for(int i = 0; i < 30; i++) {
// Inserts the events of each voice channel's silence period ending
list_insert(event_list, create_silence_period(simulation_time, 0, i));
}

// Number of packages created in each round that eventually left the system
int round_data_exits[round_number+1];
for (int i = 0; i < round_number+1; i++) round_data_exits[i] = 0;
int round_voice_exits[round_number+1];
for (int i = 0; i < round_number+1; i++) round_voice_exits[i] = 0;

//DEBUG FILES
ofstream log_file, averages_file;

```

```

int simulation_percentage = -1;
cout << endl;

cout << "Simulation in progress... 0%\r";

// Main loop of events
for (int round = 0; round < round_number+1; round++) {
//--- Variables needed to calculate the statistics for each round ---//
double round_time = simulation_time;
//-----//


if (allow_logging) {
stringstream sstm;
sstm << "log_" << round << ".txt";
log_file.open (sstm.str().c_str());
}
// Loops until customers_number customers are sampled or, if this is round 0, until
the transient_period customers are sampled
while ((round > 0 && Customer::total_customers < customers_number * round) ||
(round == 0 && Customer::total_customers < 0)) {
Event current_event = *event_list.begin();

event_list.erase(event_list.begin());
Customer c_prev = customer_being_served; // needed to test if "treat_event" will
change the customer in the server
int data_queue_prev = data_traffic->size; // needed to test if "treat_event" will
interrupt a data package being served
int voice_queue_prev = voice_traffic->size; // both of these are needed for the Areas
Method
current_event.treat_event(round, data_traffic, voice_traffic, &customer_being_served,
preemption);
simulation_time = current_event.time;

//===== Areas Method =====//
if (data_queue_prev != data_traffic->size) {
if (time_data == 0) {
time_data = simulation_time;
size_data = data_traffic->size;
} else {
Nq1[round] += fabs((simulation_time - time_data) * (size_data));
time_data = simulation_time;
size_data = data_traffic->size;
}
}
if (voice_queue_prev != voice_traffic->size) {

```

```

if (time_voice == 0) {
    time_voice = simulation_time;
    size_voice = voice_traffic->size;
} else {
    Nq2[round] += fabs((simulation_time - time_voice) * (size_voice));
    time_voice = simulation_time;
    size_voice = voice_traffic->size;
}
}

//=====

if (current_event.etype == ARRIVAL && current_event.customer.type == DATA) {
    list_insert(event_list, create_data(-99999, simulation_time, lambda)); // next data package
} else if (current_event.etype == ARRIVAL && current_event.customer.type == VOICE){
    if (voice_channels[current_event.channel_id] > 0) {
        voice_channels[current_event.channel_id]--;
        list_insert(event_list, create_voice(-99999, simulation_time, 16,
        current_event.channel_id)); // next voice package of this channel
    } else {
        list_insert(event_list, create_silence_period(simulation_time, 16,
        current_event.channel_id)); // starts next silence period 16ms later
    }
    if (data_queue_prev < data_traffic->size) { // if a voice arrival increased the data queue,
    that means a data package was interrupted
        Customer removed_customer = queue_remove(data_traffic); // the data package that suffered
        interruption
        removed_customer.time_in_service += (simulation_time - removed_customer.checkpoint_time);
        removed_customer.checkpoint_time = simulation_time;
        queue_return(data_traffic, removed_customer);
        if (allow_logging) log_file << "1 -- " << "REMOVED " << removed_customer.id << " FROM DATA
        QUEUE" << endl;
        list_remove(event_list, removed_customer.id);
    }
}

} else if (current_event.etype == SILENCE_END) {
    if (last_time[current_event.channel_id] != 0) last_time[current_event.channel_id] = 0;
    voice_channels[current_event.channel_id] = voice_package_number();
    if (voice_channels[current_event.channel_id] > 0) {
        voice_channels[current_event.channel_id]--;
        list_insert(event_list, create_voice(-99999, simulation_time, 0,
        current_event.channel_id)); // next voice package of this channel
    }
} else if (current_event.etype == EXIT && current_event.customer.type == DATA){
    T1[current_event.customer.round] += (current_event.time -
    current_event.customer.arrival_time);
}

```

```

W1[current_event.customer.round] += current_event.customer.time_in_queue;
current_event.customer.time_in_service += (simulation_time -
current_event.customer.checkpoint_time);
X1[current_event.customer.round] += current_event.customer.time_in_service;

round_data_exits[current_event.customer.round]++;
} else if (current_event.etype == EXIT && current_event.customer.type == VOICE){
T2[current_event.customer.round] += (current_event.time -
current_event.customer.arrival_time);
W2[current_event.customer.round] += current_event.customer.time_in_queue;

round_voice_exits[current_event.customer.round]++;
}
if (customer_being_served.id != c_prev.id) { // checks if a new customer arrived at
the server due to this event
if (customer_being_served.type != NONE) {
customer_being_served.time_in_queue += (simulation_time -
customer_being_served.checkpoint_time);
customer_being_served.checkpoint_time = simulation_time;
//If a Voice is being served, gets the interval
if(customer_being_served.type == VOICE){
//If it is not the first voice package of the channel
if (last_time[customer_being_served.channel_id] != 0){
intervals.push_back(customer_being_served.checkpoint_time -
last_time[customer_being_served.channel_id]);
}
last_time[customer_being_served.channel_id] = customer_being_served.checkpoint_time;
}
list_insert(event_list, remove_package(simulation_time, customer_being_served));
}
}

// Percentage of simulation complete
if (Customer::total_customers > 0 && simulation_percentage != (int)((float)
Customer::total_customers * 100 / (customers_number * round_number))) {
cout << "Simulation in progress... " << (int)((float) Customer::total_customers * 100
/ (customers_number * round_number)) << "%" << '\r';
simulation_percentage = (int)((float) Customer::total_customers * 100 / (customers_number *
round_number));
}

//=====
if (allow_logging) {
log_file << "1 -- " << "TIME: " << current_event.time << "ms TYPE: ";
if (current_event.etype == ARRIVAL && current_event.customer.type == DATA) log_file

```

```

<< "Data Arrival";
else if (current_event.etype == ARRIVAL && current_event.customer.type == VOICE) log_file
<< "Voice Arrival";
else if (current_event.etype == SILENCE_END) log_file << "Silence End";
else if (current_event.etype == EXIT && current_event.customer.type == DATA) log_file
<< "Data Departure";
else if (current_event.etype == EXIT && current_event.customer.type == VOICE) log_file
<< "Voice Departure";
if (current_event.etype != SILENCE_END) log_file << " CUSTOMER ID: "<<
(current_event.customer.id);
log_file << " CHANNEL ID: " << current_event.channel_id << "\n";
}
//=====
}

if (allow_logging) {
log_file << "End of Round " << round << " ; Duration: " << (simulation_time - round_time)
<< "ms\n\n";
log_file.close();
}
// To be done on all rounds except for the transient period (round 0)
if (round > 0) {
// Areas Method requires dividing the area by the time spent
Nq1[round] /= (simulation_time - round_time);
Nq2[round] /= (simulation_time - round_time);

// Sum all the intervals into delta of the round
for (int i = 0; i < intervals.size(); i++){
EDelta[round] += intervals[i];
}

// If there is some interval, divide by the quantity to get the average
if (!intervals.empty()) EDelta[round] = EDelta[round]/intervals.size();

for (int i = 0; i < intervals.size(); i++){
VDelta[round] += pow((intervals[i] - EDelta[round]), 2);
}

if (intervals.size() <= 1) VDelta[round] = 0;
else VDelta[round] = VDelta[round]/(intervals.size() - 1);

intervals.clear();
}
}

```

```

// Divide the sum of times by the number of events to find the average
for (int round = 0; round < round_number+1; round++) {
    if (round_data_exits[round] > 0) {
        T1[round] /= round_data_exits[round];
        W1[round] /= round_data_exits[round];
        X1[round] /= round_data_exits[round];
    }
    if (round_voice_exits[round] > 0) {
        T2[round] /= round_voice_exits[round];
        W2[round] /= round_voice_exits[round];
    }
}

cout << endl;

// ----- Finding the averages of the confidence intervals ----- //
// Only rounds from which at least one package left the system are eligible for certain
statistics.
int eligible_data_rounds = 0, eligible_voice_rounds = 0;
for(int i=1; i < round_number+1; i++) {
    if (round_data_exits[i] > 0) eligible_data_rounds++;
    if (round_voice_exits[i] > 0) eligible_voice_rounds++;
    ET1 += T1[i];
    EW1 += W1[i];
    EX1 += X1[i];
    ENq1 += Nq1[i];
    ET2 += T2[i];
    EW2 += W2[i];
    ENq2 += Nq2[i];
    EEDelta += EDelta[i];
    EVDelta += VDelta[i];
}
ET1 /= eligible_data_rounds;
EW1 /= eligible_data_rounds;
EX1 /= eligible_data_rounds;
ENq1 /= round_number;
ET2 /= eligible_voice_rounds;
EW2 /= eligible_voice_rounds;
ENq2 /= round_number;
EEDelta /= round_number;
EVDelta /= round_number;

// Finding the standard deviations of the confidence intervals
for(int i=1; i < round_number+1; i++) {
    ST1 += pow(T1[i] - ET1, 2);
}

```

```

SW1 += pow(W1[i] - EW1, 2);
SX1 += pow(X1[i] - EX1, 2);
SNq1 += pow(Nq1[i] - ENq1, 2);
ST2 += pow(T2[i] - ET2, 2);
SW2 += pow(W2[i] - EW2, 2);
SNq2 += pow(Nq2[i] - ENq2, 2);
SEDelta += pow(EDelta[i] - EEDelta, 2);
SVDelta += pow(VDelta[i] - EVDelta, 2);
}
ST1 /= (round_number - 1);
ST1 = sqrt(ST1);
SW1 /= (round_number - 1);
SW1 = sqrt(SW1);
SX1 /= (round_number - 1);
SX1 = sqrt(SX1);
SNq1 /= (round_number - 1);
SNq1 = sqrt(SNq1);
ST2 /= (round_number - 1);
ST2 = sqrt(ST2);
SW2 /= (round_number - 1);
SW2 = sqrt(SW2);
SNq2 /= (round_number - 1);
SNq2 = sqrt(SNq2);
SEDelta /= (round_number - 1);
SEDelta = sqrt(SEDelta);
SVDelta /= (round_number - 1);
SVDelta = sqrt(SVDelta);

// Finding the confidence intervals.
// Lower limits must always be non-negative.
lower_T1 = ET1 - 1.645 * ST1 / sqrt(round_number); lower_T1 = (lower_T1 > 0) ? lower_T1 : 0;
upper_T1 = ET1 + 1.645 * ST1 / sqrt(round_number);
lower_W1 = EW1 - 1.645 * SW1 / sqrt(round_number); lower_W1 = (lower_W1 > 0) ? lower_W1 : 0;
upper_W1 = EW1 + 1.645 * SW1 / sqrt(round_number);
lower_X1 = EX1 - 1.645 * SX1 / sqrt(round_number); lower_X1 = (lower_X1 > 0) ? lower_X1 : 0;
upper_X1 = EX1 + 1.645 * SX1 / sqrt(round_number);
lower_Nq1 = ENq1 - 1.645 * SNq1 / sqrt(round_number); lower_Nq1 = (lower_Nq1 > 0) ? lower_Nq1 : 0;
upper_Nq1 = ENq1 + 1.645 * SNq1 / sqrt(round_number);
lower_T2 = ET2 - 1.645 * ST2 / sqrt(round_number); lower_T2 = (lower_T2 > 0) ? lower_T2 : 0;
upper_T2 = ET2 + 1.645 * ST2 / sqrt(round_number);
lower_W2 = EW2 - 1.645 * SW2 / sqrt(round_number); lower_W2 = (lower_W2 > 0) ? lower_W2 : 0;
upper_W2 = EW2 + 1.645 * SW2 / sqrt(round_number);
lower_Nq2 = ENq2 - 1.645 * SNq2 / sqrt(round_number); lower_Nq2 = (lower_Nq2 > 0) ? lower_Nq2 : 0;
upper_Nq2 = ENq2 + 1.645 * SNq2 / sqrt(round_number);
lower_EDelta = EEDelta - 1.645 * SEDelta / sqrt(round_number); lower_EDelta =
(lower_EDelta > 0) ? lower_EDelta : 0;
upper_EDelta = EEDelta + 1.645 * SEDelta /

```

```

sqrt(round_number);
lower_VDelta = EVDelta - 1.645 * SVDelta / sqrt(round_number); lower_VDelta =
(lower_VDelta > 0) ? lower_VDelta : 0; upper_VDelta = EVDelta + 1.645 * SVDelta /
sqrt(round_number);

// Creates a file where all the averages of each round are stored
averages_file.open ("averages.txt");
averages_file << "\nT1: [ ";
for (int i = 0; i < round_number; i++) averages_file << T1[i] << ", "; averages_file <<
T1[round_number] << " ]\n";

averages_file << "\nW1: [ ";
for (int i = 0; i < round_number; i++) averages_file << W1[i] << ", "; averages_file <<
W1[round_number] << " ]\n";

averages_file << "\nX1: [ ";
for (int i = 0; i < round_number; i++) averages_file << X1[i] << ", "; averages_file <<
X1[round_number] << " ]\n";

averages_file << "\nNq1: [ ";
for (int i = 0; i < round_number; i++) averages_file << Nq1[i] << ", "; averages_file <<
Nq1[round_number] << " ]\n";

averages_file << "\nT2: [ ";
for (int i = 0; i < round_number; i++) averages_file << T2[i] << ", "; averages_file <<
T2[round_number] << " ]\n";

averages_file << "\nW2: [ ";
for (int i = 0; i < round_number; i++) averages_file << W2[i] << ", "; averages_file <<
W2[round_number] << " ]\n";

averages_file << "\nNq2: [ ";
for (int i = 0; i < round_number; i++) averages_file << Nq2[i] << ", "; averages_file <<
Nq2[round_number] << " ]\n";

averages_file << "\nE[Delta]: [ ";
for (int i = 0; i < round_number; i++) averages_file << EDelta[i] << ", "; averages_file <<
EDelta[round_number] << " ]\n";

averages_file << "\nV[Delta]: [ ";
for (int i = 0; i < round_number; i++) averages_file << VDelta[i] << ", "; averages_file <<
VDelta[round_number] << " ]\n";

averages_file.close();

```

```

// Prints the results
cout << "Confidence intervals: " << endl;
cout << "\nE[T1]: " << lower_T1 << " < " << ET1 << " < " << upper_T1 << endl;
cout << "\nE[W1]: " << lower_W1 << " < " << EW1 << " < " << upper_W1 << endl;
cout << "\nE[X1]: " << lower_X1 << " < " << EX1 << " < " << upper_X1 << endl;
cout << "\nE[Nq1]: " << lower_Nq1 << " < " << ENq1 << " < " << upper_Nq1 << endl;
cout << "\nE[T2]: " << lower_T2 << " < " << ET2 << " < " << upper_T2 << endl;
cout << "\nE[W2]: " << lower_W2 << " < " << EW2 << " < " << upper_W2 << endl;
cout << "\nE[Nq2]: " << lower_Nq2 << " < " << ENq2 << " < " << upper_Nq2 << endl;
cout << "\nE[Delta]: " << lower_EDelta << " < " << EEDelta << " < " << upper_EDelta << endl;
cout << "\nV[Delta]: " << lower_VDelta << " < " << EVDelta << " < " << upper_VDelta << endl;

// Print the data necessary for the graph plotting program into a file
stringstream sstm;
ofstream graph_data;
sstm << "graph_data" << seed << ".txt";
graph_data.open (sstm.str().c_str());
for (int i=1; i < round_number+1; i++) {
graph_data << i << "," << T1[i] << "," << W1[i] << "," << X1[i] << "," << Nq1[i] << ","
<< T2[i] << "," << W2[i] << "," << Nq2[i] << "," << EDelta[i] << "," << VDelta[i] << endl;
}
graph_data.close();

}

void execution(unsigned int seed, int transient_period, int customers_number, int
round_number, float utilization1, bool preemption, bool allow_logging){
    // The service 1 average time is going to be the package size average divided by the
    transmission rate
    float service_average1 = (float) (755 * 8) / (float) (0.002 * 1024 * 1024);
    float lambda = utilization1 / service_average1;
    // Starts rounds
    rounds(seed, transient_period, customers_number, round_number, lambda, preemption,
allow_logging);
}

```

4.3 Customer.cpp

```

#include "Customer.h"

// Sets the total of customers to zero
int Customer::total_customers = 0;

```

```

Customer::Customer() {
    this->id = -99999;
    this->round = -99999;
    this->type = NONE;
    this->size = 0;
    this->service_time = 0;
    this->time_in_queue = 0;
    this->time_in_service = 0;
    this->arrival_time = 0;
    this->checkpoint_time = 0;
}

// Defines a Customer with identification
Customer::Customer(int id, int round, customer_type type, double arrival_time,
int channel_id) {
    this->id = id;
    this->round = round;
    this->type = type;
    if (type == DATA) this->size = data_package_size() * 8;
    else if (type == VOICE) this->size = 512;
    this->service_time = (double) this->size / (0.002 * 1024 * 1024);
    this->time_in_queue = 0;
    this->time_in_service = 0;
    this->arrival_time = arrival_time;
    this->checkpoint_time = arrival_time;
    this->channel_id = channel_id;
}

// Function that determines the size of a data package in bytes (actual function)
int data_package_size() {
    long double seed = rand1();
    if (seed < .3) return 64;
    else if (seed < .3 + .1) return 512;
    else if (seed < .3 + .1 + .3) return 1500;
    else {
        seed = (seed - .3 - .1 - .3) / (1 - .3 - .1 - .3);
        return seed * (1500 - 64) + 64;
    }
}

```

4.4 Customer.h

```

#ifndef CUSTOMER_H
#define CUSTOMER_H

#ifndef STATISTICS_H

```

```

#include "statistics.cpp"
#endif

// Defines the type of the customer
typedef enum {
NONE, // "phantom" customer, returned when dequeuing an empty queue
DATA,
VOICE
} customer_type;

class Customer {
public:
int id;
int round; // the round in which this customer joined the queue
double arrival_time; // when this customer joined the queue
double checkpoint_time; // used to know when was the last time this customer was sent to
the queue/server
customer_type type;
static int total_customers; // used for giving the next customer a new id
double service_time; // how long it takes the server to serve this customer
double time_in_service; // X1
double time_in_queue; // W1
int size; // in bits
int channel_id; // the id of the voice channel related to the customer
// Customer(customer_type type, double arrival_time);
Customer();
Customer(int id, int round, customer_type type, double arrival_time, int channel_id = -1);
};

// Function that determines the size of a data package in bytes (actual function)
int data_package_size();

#endif

```

4.5 data_structures.cpp

```

#include "data_structures.h"
#ifndef STDLIB
#include <stdlib.h>
#define STDLIB
#endif
#ifndef LIST
#include <list>
#define LIST
#endif

```

```

// Returns a new queue with size 0
queue* queue_create() {
    queue* q = (queue*) malloc(sizeof(queue));
    q->size = 0;
    q->head_of_line = NULL;
    q->end_of_line = NULL;
    return q;
}

// Inserts customer "c" at the end of a queue "q"
void queue_insert(queue *q, Customer c) {
    if (q->end_of_line == NULL) {
        q->end_of_line = (linked_list*) malloc(sizeof(linked_list));
        q->end_of_line->customer = c;
        q->end_of_line->next = NULL;
        q->end_of_line->previous = NULL;
        q->head_of_line = q->end_of_line;
    } else {
        q->end_of_line->next = (linked_list*) malloc(sizeof(linked_list));
        q->end_of_line->next->customer = c;
        q->end_of_line->next->next = NULL;
        q->end_of_line->next->previous = q->end_of_line;
        q->end_of_line = q->end_of_line->next;
    }
    q->size++;
}

// Returns customer "c" back to the head of a queue "q"
void queue_return(queue *q, Customer c) {
    if (q->head_of_line == NULL) {
        q->head_of_line = (linked_list*) malloc(sizeof(linked_list));
        q->head_of_line->customer = c;
        q->head_of_line->next = NULL;
        q->head_of_line->previous = NULL;
        q->end_of_line = q->head_of_line;
    } else {
        q->head_of_line->previous = (linked_list*) malloc(sizeof(linked_list));
        q->head_of_line->previous->customer = c;
        q->head_of_line->previous->previous = NULL;
        q->head_of_line->previous->next = q->head_of_line;
        q->head_of_line = q->head_of_line->previous;
    }
    q->size++;
}

// Removes the head of the line and returns it

```

```

// If queue is empty, returns customer of type NONE
Customer queue_remove(queue *q) {
if (q->head_of_line == NULL) {
return Customer();
} else {
Customer ret = q->head_of_line->customer;
linked_list *curr_head = q->head_of_line;
q->head_of_line = q->head_of_line->next;
free(curr_head);
q->size--;
if (q->size == 0) q->end_of_line = NULL;
return ret;
}
}

/*
// Prints the queue
void queue_print(queue *q) {
linked_list *pointer = q->head_of_line;
while (pointer != q->end_of_line) {
cout << pointer->customer.size << "\n";
pointer = pointer->next;
}
if (q->end_of_line != NULL) cout << q->end_of_line->customer.size;
}
*/

```

4.6 data structures.h

```

#ifndef DATA_STRUCTURES_H
#define DATA_STRUCTURES_H

#ifndef CUSTOMER_H
#include "Customer.cpp"
#endif

// Sets the linked list
typedef struct LINKED_LIST {
Customer customer;
struct LINKED_LIST *next;
struct LINKED_LIST *previous;
} linked_list;

// Sets the structure of a queue
typedef struct QUEUE {
int size;

```

```

linked_list *head_of_line;
linked_list *end_of_line;
} queue;

queue* queue_create();
void queue_insert(queue *q, Customer c);
void queue_return(queue *q, Customer c);
Customer queue_remove(queue *q);

#endif

```

4.7 Event.cpp

```

#include "Event.h"
#ifndef IOSTREAM
#include <iostream>
#define IOSTREAM
#endif
#ifndef LIST
#include <list>
#define LIST
#endif
using namespace std;

// Constructor for events related to data/voice packages. Channel id means the channel
that originated
// the voice package. Data packages may leave it at -1 as default.
Event::Event(double time, Customer c, event_type etype, int channel_id) {
this->time = time;
this->customer = c;
this->channel_id = channel_id;
this->etype = etype;
}

// Constructor for events related to voice channels.
Event::Event(double time, int channel_id) {
this->time = time;
this->customer = Customer();
this->channel_id = channel_id;
this->etype = SILENCE_END;
}

// Method that updates the system's queues and server based on this event's information
void Event::treat_event(queue *data_queue, queue *voice_queue, Customer *current, bool
preemption) {
if (this->etype == ARRIVAL) {

```

```

this->customer.id = Customer::total_customers++;
if (current->type == NONE) {
*current = Customer(this->customer.id, this->customer.round, this->customer.type,
this->time, this->customer.channel_id);
} else if (this->customer.type == DATA) {
queue_insert(data_queue, Customer(this->customer.id, this->customer.round, DATA,
this->time));
} else if (this->customer.type == VOICE) {
if (preemption) {
if (current->type == DATA) {
queue_return(data_queue, *current);
*current = Customer(Customer(this->customer.id, this->customer.round, VOICE,
this->time, this->customer.channel_id));
} else queue_insert(voice_queue, Customer(this->customer.id, this->customer.round,
VOICE, this->time, this->customer.channel_id));
} else {
queue_insert(voice_queue, Customer(this->customer.id, this->customer.round, VOICE,
this->time, this->customer.channel_id));
}
}
} else if (this->etype == EXIT) {
if (voice_queue->size > 0) {
*current = queue_remove(voice_queue);
} else if (data_queue->size > 0) {
*current = queue_remove(data_queue);
} else *current = Customer();
} else {} // Treat silence period end
}

// Inserts a new event into the event list in chronological order
void list_insert(list<Event> &event_list, Event event) {
for (list<Event>::iterator it = event_list.begin(); it != event_list.end(); it++) {
if (it->time > event.time) {
//cout << "arrival time: " << event.time << "\n";
event_list.insert(it, event);
return;
}
}
event_list.insert(event_list.end(), event);
}

// Removes an event from the event list, based on the corresponding customer's id
// Events related to voice channels should be removed with
event_list.erase(event_list.begin())
void list_remove(list<Event> &event_list, int customer_id) {
for (list<Event>::iterator it = event_list.begin(); it != event_list.end(); it++) {

```

```

if (it->customer.id == customer_id) {
    event_list.erase(it);
    return;
}
}
}
}

```

4.8 Event.h

```

#ifndef EVENT_H
#define EVENT_H

#ifndef LIST
#include <list>
#define LIST
#endif
using namespace std;
#ifndef DATA_STRUCTURES_H
#include "data_structures.cpp"
#endif
/*
Only 3 types of event are necessary.
When a DATA package arrives, it's added to the data queue.
When a VOICE package arrives, it's added to the voice queue OR interrupts the data being served.
When a DATA package exits, the next voice package in line is served (if any) OR the next data package in line is served (if any) OR the server becomes empty.
When a VOICE package exits, the next voice package in line is served (if any) OR the next data package in line is served (if any) OR the server becomes empty.
When the SILENCE period ends, the corresponding voice channel generates the voice package arrival events.
*/
// Defines the type of the event
typedef enum {
    ARRIVAL,
    EXIT,
    SILENCE_END
} event_type;

class Event {
public:
    double time; // the time when it begins
    Customer customer; // the customer related to the event
    int channel_id; // the id of the voice channel related to the event
    event_type etype; // if the event is an arrival in the system or an exit of it
}

```

```

Event(double time, Customer c, event_type etype, int channel_id = -1);
Event(double time, int channel_id);
void treat_event(queue *data_queue, queue *voice_queue, Customer *current, bool
preemption);
};

// Inserts a new event into the event list in chronological order
void list_insert(list<Event> *event_list, Event event);
// Removes an event from the event list, based on the corresponding customer's id
// Events related to voice channels should be removed with
event_list.erase(event_list.begin())
void list_remove(list<Event> *event_list, int customer_id);

#endif

```

4.9 statistics.cpp

```

#include "statistics.h"
#include <iostream>
using namespace std;
#ifndef STDLIB
#include <stdlib.h>
#define STDLIB
#endif
#ifndef MATH
#include <math.h>
#define MATH
#endif

// Function that generates a random number between 0 and 1
long double rand1() {
    return (long double) rand() / (long double) RAND_MAX;
}

// Sample Mean function:
// Gives the mean of a sample of BIGNUM random values generated by the given function.

// Discrete variable with no parameter
double sample_mean(int (*function)()) {
    double ret = 0;
    int current_sample;
    for (int i = 0; i < BIGNUM; i++) {
        current_sample = (*function)();
        ret += current_sample;
    }
    return ret / BIGNUM;
}

```

```

}

// Discrete variable with integer parameter
double sample_mean(int (*function)(int), int i) {
double ret = 0;
int current_sample;
for (int i = 0; i < BIGNUM; i++) {
current_sample = (*function)(i);
ret += current_sample;
}
return ret / BIGNUM;
}

// Continuous variable with no parameter
double sample_mean(double (*function)()) {
double ret = 0;
double current_sample;
for (int i = 0; i < BIGNUM; i++) {
current_sample = (*function)();
ret += current_sample;
}
return ret / BIGNUM;
}

// Continuous variable with floating point parameter
long double sample_mean(double (*function)(double), double d) {
double ret = 0;
double current_sample;
for (int i = 0; i < BIGNUM; i++) {
current_sample = (*function)(d);
ret += current_sample;
}
return ret / BIGNUM;
}

// Generates a random number with an exponential distribution
double exponential(double rate) {
long double seed = rand1();
if (seed < .000001) seed = .000001; // seed cannot be 0 or there would be a log(0), which
is infinite, so... we make it a little smaller
//cout << "lambda: " << rate << " arrival time: " << -log(seed) / rate << "\n";
return -log(seed) / rate;
}

```

4.10 statistics.h

```
#ifndef STATISTICS_H
#define STATISTICS_H

// Number used for large sample sizes
#define BIGNUM 2000000

// Function that generates a random number between 0 and 1
long double rand1();

// Sample Mean function:
// Gives the mean of a sample of BIGNUM random values generated by the given function.

// Discrete variable with no parameter
double sample_mean(int (*function)());

// Discrete variable with integer parameter
double sample_mean(int (*function)(int), int i);

// Continuous variable with no parameter
double sample_mean(double (*function)());

// Continuous variable with floating point parameter
long double sample_mean(double (*function)(double), double d);

// Generates a random number with an exponential distribution
double exponential(double rate);

#endif
```

4.11 test correction.cpp

```
#include "run_queues.cpp"
#ifndef STDIO
#include <stdio.h>
#define STDIO
#endif
#ifndef STDLIB
#include <stdlib.h>
#define STDLIB
#endif
#ifndef MATH
#include <math.h>
#define MATH
#endif
```

```

// Creates the arrival event of a data customer with deterministic time
Event create_data_deterministic(int customer_id, int round, float simulation_time, float lambda){
    double arrivalTime = 60;
    Customer customer = Customer(customer_id, round, DATA, simulation_time + arrivalTime);
    Event event = Event(simulation_time + arrivalTime, customer, ARRIVAL);
    return event;
}

// Creates the silence period event with deterministic time
Event create_silence_period_deterministic(float simulation_time, float offset, int channel_id){
    double arrivalTime = 100 + (channel_id * 5);
    Event event = Event(simulation_time + arrivalTime + offset, channel_id);
    return event;
}

// Creates the silence period event with exponential time
Event create_silence_period_random(float simulation_time, float offset, int channel_id){
    double arrivalTime = exponential(1.0/650);
    Event event = Event(simulation_time + arrivalTime + offset, channel_id);
    return event;
}

// It runs the rounds of the simulation for deterministic silence period if has_data is
// false and for deterministic data otherwise
void round_test(int transiente_period, int customers_number, int round_number, float lambda, bool preemption, bool allow_logging, bool has_data){
    queue* data_traffic = queue_create(); // Queue where the data packages are stored
    queue* voice_traffic = queue_create(); // Queue where the voice packages are stored

    double simulation_time = 0; // Current time in the simulator
    double last_time[30]; //Keeps the last time a package of each specific channel entered
    the server

    double total_time = 0;

    for (int i = 0; i < 30; i++){
        last_time[i] = 0;
    }

    // Gets the intervals of each channel;
    vector <double> intervals;

```

```

// Since we must ignore the first transiente_period customers, only customers considered
will be the ones with id >= 0
// and we'll start counting from -transiente_period
Customer::total_customers = -transiente_period;

Customer customer_being_served = Customer(); // The customer currently in the server.
NONE type = no customer there.

    // Expectations/Averages
float T1[round_number+1]; // Average time a data package stays in the system
float W1[round_number+1]; // Average time a data package stays in the queue
float X1[round_number+1]; // Average time a data package stays in the server
    float Nq1[round_number+1]; // Average number of data packages in the queue
float T2[round_number+1]; // Average time a voice package stays in the system
float W2[round_number+1]; // Average time a voice package stays in the queue
    float Nq2[round_number+1]; // Average number of voice packages in the queue

// Interval between packages Estimator
    float EDelta[round_number+1];
    float VDelta[round_number+1];

for (int i = 0; i < round_number+1; i++) {
T1[i] = 0;
W1[i] = 0;
X1[i] = 0;
Nq1[i] = 0;
T2[i] = 0;
W2[i] = 0;
Nq2[i] = 0;

EDelta[i] = 0;
VDelta[i] = 0;
}

// Averages, lower limits and upper limits of the confidence intervals
float ET1 = 0, lower_T1, upper_T1;
float EW1 = 0, lower_W1, upper_W1;
float EX1 = 0, lower_X1, upper_X1;
float ENq1 = 0, lower_Nq1, upper_Nq1;
float ET2 = 0, lower_T2, upper_T2;
float EW2 = 0, lower_W2, upper_W2;
float ENq2 = 0, lower_Nq2, upper_Nq2;
float EEDelta = 0, lower_EDelta, upper_EDelta;
float EVDelta = 0, lower_VDelta, upper_VDelta;

// Standard deviations of the confidence intervals

```

```

float ST1 = 0;
float SW1 = 0;
float SX1 = 0;
float SNq1 = 0;
float ST2 = 0;
float SW2 = 0;
float SNq2 = 0;
float SEDelta = 0;
float SVDelta = 0;

// Variables used for the areas method (Data Queue)
double time_data = 0; // data queue timestamps
int size_data; // data queue sizes

// Variables used for the areas method (Voice Queue)
double time_voice = 0; // voice queue timestamps
int size_voice; // voice queue sizes

// People that came out of the system coming from both Queues;
int out1 = 0; // data packages
int out2 = 0; // voice packages
int out = 0; // out1 + out2

// How many voice packages each channel needs to send before entering silence period.
int voice_channels[30];
for (int i = 0; i < 30; i++) voice_channels[i] = 0;

list<Event> event_list;

// If this is the deterministic data test, than the data package will be created
if (has_data) list_insert(event_list,
create_data_deterministic(Customer::total_customers++, 0, simulation_time, lambda));

// VOICE CHANNELS
for(int i = 0; i < 30; i++) {
    if (!has_data) list_insert(event_list,
    create_silence_period_deterministic(simulation_time, 0, i));
    else list_insert(event_list, create_silence_period_random(simulation_time, 0, i));
}

// Number of packages created in each round that eventually left the system
int round_data_exits[round_number];
for (int i = 0; i < round_number; i++) round_data_exits[i] = 0;
int round_voice_exits[round_number];
for (int i = 0; i < round_number; i++) round_voice_exits[i] = 0;

```

```

//DEBUG FILES
ofstream log_file, averages_file;
if (allow_logging) {
    // Determines each file for both tests
    if (!has_data) log_file.open ("log_deterministic_test.txt");
    else log_file.open ("log_deterministic_data_test.txt");
}

int simulation_percentage = -1;
cout << endl;

cout << "Simulation in progress... 0%\r";

// Main loop of events
for (int round = 0; round < round_number+1; round++) {
    //--- Variables needed to calculate the statistics for each round ---//
    double round_time = simulation_time;
    //-----//


    if (allow_logging) {
        stringstream sstm;
        sstm << "log_" << round << ".txt";
        log_file.open (sstm.str().c_str());
    }
    // Loops until customers_number customers are sampled or, if this is round 0, until the
    transiente_period customers are sampled
    while ((round > 0 && Customer::total_customers < customers_number * (round+1)) ||
           (round == 0 && Customer::total_customers < 0)) {
        Event current_event = *event_list.begin();

        event_list.erase(event_list.begin());
        Customer c_prev = customer_being_served; // needed to test if "treat_event" will change
        the customer in the server
        int data_queue_prev = data_traffic->size; // needed to test if "treat_event" will
        interrupt a data package being served
        int voice_queue_prev = voice_traffic->size; // both of these are needed for the Areas
        Method
        current_event.treat_event(data_traffic, voice_traffic, &customer_being_served, preemption);
        simulation_time = current_event.time;

        //===== Areas Method =====//
        if (data_queue_prev != data_traffic->size) {
            if (time_data == 0) {
                time_data = simulation_time;
                size_data = data_traffic->size;

```

```

} else {
Nq1[round] += fabs((simulation_time - time_data) * (size_data));
time_data = simulation_time;
size_data = data_traffic->size;
}
}

if (voice_queue_prev != voice_traffic->size) {
if (time_voice == 0) {
time_voice = simulation_time;
size_voice = voice_traffic->size;
} else {
Nq2[round] += fabs((simulation_time - time_voice) * (size_voice));
time_voice = simulation_time;
size_voice = voice_traffic->size;
}
}
//=====

if (current_event.etype == ARRIVAL && current_event.customer.type == DATA) {
list_insert(event_list, create_data_deterministic(-99999, round, simulation_time, lambda));
// next data package
} else if (current_event.etype == ARRIVAL && current_event.customer.type == VOICE){
if (voice_channels[current_event.channel_id] > 0) {
voice_channels[current_event.channel_id]--;
list_insert(event_list, create_voice(-99999, round, simulation_time,
16, current_event.channel_id)); // next voice package of this channel
} else {
if (!has_data) list_insert(event_list, create_silence_period_deterministic(simulation_time,
current_event.channel_id)); // starts next silence period 16ms later
else list_insert(event_list, create_silence_period_random(simulation_time, 16,
current_event.channel_id));
}
if (data_queue_prev < data_traffic->size) { // if a voice arrival increased the data queue,
that means a data package was interrupted
Customer removed_customer = queue_remove(data_traffic); // the data package that suffered
interruption
removed_customer.time_in_service += (simulation_time - removed_customer.checkpoint_time);
removed_customer.checkpoint_time = simulation_time;
queue_return(data_traffic, removed_customer);
if (allow_logging) log_file << "1 -- " << "REMOVED " << removed_customer.id << " FROM DATA
QUEUE" << endl;
list_remove(event_list, removed_customer.id);
}

} else if (current_event.etype == SILENCE_END) {

```

```

if (last_time[current_event.channel_id] != 0) last_time[current_event.channel_id] = 0;
voice_channels[current_event.channel_id] = voice_package_number();
if (voice_channels[current_event.channel_id] > 0) {
    voice_channels[current_event.channel_id]--;
    list_insert(event_list, create_voice(-99999, round, simulation_time, 0,
                                         current_event.channel_id)); // next voice package of this channel
}
} else if (current_event.etype == EXIT && current_event.customer.type == DATA){
    T1[current_event.customer.round] += (current_event.time -
                                         current_event.customer.arrival_time);
    W1[current_event.customer.round] += current_event.customer.time_in_queue;
    current_event.customer.time_in_service += (simulation_time -
                                                current_event.customer.checkpoint_time);
    X1[current_event.customer.round] += current_event.customer.time_in_service;

    round_data_exits[current_event.customer.round]++;
}
} else if (current_event.etype == EXIT && current_event.customer.type == VOICE){
    T2[current_event.customer.round] += (current_event.time -
                                         current_event.customer.arrival_time);
    W2[current_event.customer.round] += current_event.customer.time_in_queue;

    round_voice_exits[current_event.customer.round]++;
}
if (customer_being_served.id != c_prev.id) { // checks if a new customer arrived at the
server due to this event
    if (customer_being_served.type != NONE) {
        customer_being_served.time_in_queue += (simulation_time -
                                                 customer_being_served.checkpoint_time);
        customer_being_served.checkpoint_time = simulation_time;
        //If a Voice is being served, gets the interval
        if(customer_being_served.type == VOICE){
            //If it is not the first voice package of the channel
            if (last_time[customer_being_served.channel_id] != 0){
                intervals.push_back(customer_being_served.checkpoint_time -
                                     last_time[customer_being_served.channel_id]);
            }
            last_time[customer_being_served.channel_id] = customer_being_served.checkpoint_time;
        }
        list_insert(event_list, remove_package(simulation_time, customer_being_served));
    }
}

// Percentage of simulation complete
if (Customer::total_customers > 0 && simulation_percentage != (int)((float)
Customer::total_customers * 100 / (customers_number * round_number))) {

```

```

cout << "Simulation in progress... " << (int)((float) Customer::total_customers * 100 /
(customers_number * round_number)) << "%" << '\r';
simulation_percentage = (int)((float) Customer::total_customers * 100 / (customers_number *
round_number));
}

//=====
if (allow_logging) {
log_file << "1 -- " << "TIME: " << current_event.time << "ms TYPE: ";
if (current_event.etype == ARRIVAL && current_event.customer.type == DATA) log_file <<
"Data Arrival";
else if (current_event.etype == ARRIVAL && current_event.customer.type == VOICE) log_file <<
"Voice Arrival";
else if (current_event.etype == SILENCE_END) log_file << "Silence End";
else if (current_event.etype == EXIT && current_event.customer.type == DATA) log_file <<
"Data Departure";
else if (current_event.etype == EXIT && current_event.customer.type == VOICE) log_file <<
"Voice Departure";
if (current_event.etype != SILENCE_END) log_file << " CUSTOMER ID: "<<
(current_event.customer.id);
log_file << " CHANNEL ID: " << current_event.channel_id << "\n";
}
//=====

}

if (allow_logging) {
log_file << "End of Round " << round << " ; Duration: " << (simulation_time - round_time)
<< "ms\n\n";
log_file.close();
}
// Areas Method requires dividing the area by the time spent
Nq1[round] /= (simulation_time - round_time);
Nq2[round] /= (simulation_time - round_time);

// Sum all the intervals into delta of the round
for (int i = 0; i < intervals.size(); i++){
EDelta[round] += intervals[i];
}

// If there is some interval, divide by the quantity to get the average
if (!intervals.empty()) EDelta[round] = EDelta[round]/intervals.size();

for (int i = 0; i < intervals.size(); i++){
VDelta[round] += pow((intervals[i] - EDelta[round]), 2);
}

```

```

if (intervals.size() <= 1) VDelta[round] = 0;
else VDelta[round] = VDelta[round]/(intervals.size() - 1);

intervals.clear();

}

// Divide the sum of times by the number of events to find the average
for (int round = 0; round < round_number+1; round++) {
if (round_data_exits[round] > 0) {
T1[round] /= round_data_exits[round];
W1[round] /= round_data_exits[round];
X1[round] /= round_data_exits[round];
}
if (round_voice_exits[round] > 0) {
T2[round] /= round_voice_exits[round];
W2[round] /= round_voice_exits[round];
}
}

cout << endl;

// ----- Finding the averages of the confidence intervals ----- //
// Only rounds from which at least one package left the system are eligible for certain
statistics.
int eligible_data_rounds = 0, eligible_voice_rounds = 0;
for(int i=1; i < round_number+1; i++) {
if (round_data_exits[i] > 0) eligible_data_rounds++;
if (round_voice_exits[i] > 0) eligible_voice_rounds++;
ET1 += T1[i];
EW1 += W1[i];
EX1 += X1[i];
ENq1 += Nq1[i];
ET2 += T2[i];
EW2 += W2[i];
ENq2 += Nq2[i];
EEDelta += EDelta[i];
EVDelta += VDelta[i];
}
ET1 /= eligible_data_rounds;
EW1 /= eligible_data_rounds;
EX1 /= eligible_data_rounds;
ENq1 /= round_number;
ET2 /= eligible_voice_rounds;
EW2 /= eligible_voice_rounds;

```

```

ENq2 /= round_number;
EEDelta /= round_number;
EVDelta /= round_number;

// Finding the standard deviations of the confidence intervals
for(int i=1; i < round_number+1; i++) {
    ST1 += pow(T1[i] - ET1, 2);
    SW1 += pow(W1[i] - EW1, 2);
    SX1 += pow(X1[i] - EX1, 2);
    SNq1 += pow(Nq1[i] - ENq1, 2);
    ST2 += pow(T2[i] - ET2, 2);
    SW2 += pow(W2[i] - EW2, 2);
    SNq2 += pow(Nq2[i] - ENq2, 2);
    SEDelta += pow(EDelta[i] - EEDelta, 2);
    SVDelta += pow(VDelta[i] - EVDelta, 2);
}
ST1 /= (round_number - 1);
ST1 = sqrt(ST1);
SW1 /= (round_number - 1);
SW1 = sqrt(SW1);
SX1 /= (round_number - 1);
SX1 = sqrt(SX1);
SNq1 /= (round_number - 1);
SNq1 = sqrt(SNq1);
ST2 /= (round_number - 1);
ST2 = sqrt(ST2);
SW2 /= (round_number - 1);
SW2 = sqrt(SW2);
SNq2 /= (round_number - 1);
SNq2 = sqrt(SNq2);
SEDelta /= (round_number - 1);
SEDelta = sqrt(SEDelta);
SVDelta /= (round_number - 1);
SVDelta = sqrt(SVDelta);

// Finding the confidence intervals.
// Lower limits must always be non-negative.
lower_T1 = ET1 - 1.645 * ST1 / sqrt(round_number); lower_T1 = (lower_T1 > 0) ?
lower_T1 : 0; upper_T1 = ET1 + 1.645 * ST1 / sqrt(round_number);
lower_W1 = EW1 - 1.645 * SW1 / sqrt(round_number); lower_W1 = (lower_W1 > 0) ?
lower_W1 : 0; upper_W1 = EW1 + 1.645 * SW1 / sqrt(round_number);
lower_X1 = EX1 - 1.645 * SX1 / sqrt(round_number); lower_X1 = (lower_X1 > 0) ?
lower_X1 : 0; upper_X1 = EX1 + 1.645 * SX1 / sqrt(round_number);
lower_Nq1 = ENq1 - 1.645 * SNq1 / sqrt(round_number); lower_Nq1 = (lower_Nq1 > 0) ?
lower_Nq1 : 0; upper_Nq1 = ENq1 + 1.645 * SNq1 / sqrt(round_number);
lower_T2 = ET2 - 1.645 * ST2 / sqrt(round_number); lower_T2 = (lower_T2 > 0) ?

```

```

lower_T2 : 0; upper_T2 = ET2 + 1.645 * ST2 / sqrt(round_number);
lower_W2 = EW2 - 1.645 * SW2 / sqrt(round_number); lower_W2 = (lower_W2 > 0) ?
lower_W2 : 0; upper_W2 = EW2 + 1.645 * SW2 / sqrt(round_number);
lower_Nq2 = ENq2 - 1.645 * SNq2 / sqrt(round_number); lower_Nq2 = (lower_Nq2 > 0) ?
lower_Nq2 : 0; upper_Nq2 = ENq2 + 1.645 * SNq2 / sqrt(round_number);
lower_EDelta = EEDelta - 1.645 * SEDelta / sqrt(round_number); lower_EDelta =
(lower_EDelta > 0) ? lower_EDelta : 0; upper_EDelta = EEDelta + 1.645 * SEDelta /
sqrt(round_number);
lower_VDelta = EVDelta - 1.645 * SVDelta / sqrt(round_number); lower_VDelta =
(lower_VDelta > 0) ? lower_VDelta : 0; upper_VDelta = EVDelta + 1.645 * SVDelta /
sqrt(round_number);

// Creates a file where all the averages of each round are stored
if (allow_logging) {
    if (!has_data) averages_file.open ("averages_deterministic_test.txt");
    else {
        averages_file.open ("averages_deterministic_data_test.txt");
        averages_file << "\nT1: [ ";
        for (int i = 0; i < round_number-1; i++) averages_file << T1[i] << ", ";
        averages_file << T1[round_number-1] << " ]\n";

        averages_file << "\nW1: [ ";
        for (int i = 0; i < round_number-1; i++) averages_file << W1[i] << ", ";
        averages_file << W1[round_number-1] << " ]\n";

        averages_file << "\nX1: [ ";
        for (int i = 0; i < round_number-1; i++) averages_file << X1[i] << ", ";
        averages_file << X1[round_number-1] << " ]\n";

        averages_file << "\nNq1: [ ";
        for (int i = 0; i < round_number-1; i++) averages_file << Nq1[i] << ", ";
        averages_file << Nq1[round_number-1] << " ]\n";
    }

    averages_file << "\nT2: [ ";
    for (int i = 0; i < round_number-1; i++) averages_file << T2[i] << ", ";
    averages_file << T2[round_number-1] << " ]\n";

    averages_file << "\nW2: [ ";
    for (int i = 0; i < round_number-1; i++) averages_file << W2[i] << ", ";
    averages_file << W2[round_number-1] << " ]\n";

    averages_file << "\nNq2: [ ";
    for (int i = 0; i < round_number-1; i++) averages_file << Nq2[i] << ", ";
    averages_file << Nq2[round_number-1] << " ]\n";
}

```

```

averages_file << "\nE[Delta]: [ ";
for (int i = 0; i < round_number-1; i++) averages_file << EDelta[i] << ", ";
averages_file << EDelta[round_number-1] << " ]\n";

averages_file << "\nV[Delta]: [ ";
for (int i = 0; i < round_number-1; i++) averages_file << VDelta[i] << ", ";
averages_file << VDelta[round_number-1] << " ]\n";

averages_file.close();
}

// Prints the results
cout << "Intervalos de confianca: " << endl;
if (has_data){
    cout << "\nE[T1]: " << lower_T1 << " < " << ET1 << " < " << upper_T1 << endl;
    cout << "\nE[W1]: " << lower_W1 << " < " << EW1 << " < " << upper_W1 << endl;
    cout << "\nE[X1]: " << lower_X1 << " < " << EX1 << " < " << upper_X1 << endl;
    cout << "\nE[Nq1]: " << lower_Nq1 << " < " << ENq1 << " < " << upper_Nq1 << endl;
}
cout << "\nE[T2]: " << lower_T2 << " < " << ET2 << " < " << upper_T2 << endl;
cout << "\nE[W2]: " << lower_W2 << " < " << EW2 << " < " << upper_W2 << endl;
cout << "\nE[Nq2]: " << lower_Nq2 << " < " << ENq2 << " < " << upper_Nq2 << endl;
cout << "\nE[Delta]: " << lower_EDelta << " < " << EEDelta << " < " << upper_EDelta << endl;
cout << "\nV[Delta]: " << lower_VDelta << " < " << EVDelta << " < " << upper_VDelta << endl;
}

// Tests the queue about receiving new packages
void queue_test(){
queue *data_queue = queue_create();
queue *voice_queue = queue_create();
char input = '\0';
int timestamp = 0;

cout << "COMMANDS:\n\tD = New data package\n\tV = New voice package\n\tP = Print next\n\tE or Q = Exit test\n\n";

while (input != 'e' && input != 'E' && input != 'q' && input != 'Q') {
    cout << "Data packages: " << data_queue->size << "\n";
    cout << "Voice packages: " << voice_queue->size << "\n";
    cout << "\n";
    cin >> input;
    if (input == 'D' || input == 'd') {

```

```

Customer c = Customer(0, 0, DATA, timestamp++);
queue_insert(data_queue, c);
cout << "Inserted data package! (size = " << c.size << " bits)\n";
} else if (input == 'V' || input == 'v') {
Customer c = Customer(0, 0, VOICE, timestamp++);
queue_insert(voice_queue, c);
cout << "Inserted voice package! (size = " << c.size << " bits)\n";
} else if (input == 'P' || input == 'p') {
if (voice_queue->size > 0) {
Customer c = queue_remove(voice_queue);
cout << "Type: " << c.type << "\nSize: " << c.size << "\n\n";
}
else if (data_queue->size > 0) {
Customer c = queue_remove(data_queue);
cout << "Type: " << c.type << "\nSize: " << c.size << "\n\n";
}
}
cout << "COMMANDS:\n\tD = New data package\n\tV = New voice package\n\tP = Print next\n\tE or Q = Exit test\n\n";
}

cout << '\n';
}

// Tests if the insertion and removal in the list are right
void event_list_test(){
list<Event> event_list;
char input;
double input2;
int input3;
int id = 0;

cout << "COMMANDS:\n\tP = Print event list\n\tI = Insert event\n\tR = Remove event\n\tE or Q = Exit test\n\n";

while(input != 'e' && input != 'E' && input != 'q' && input != 'Q') {
cout << "\n";
cin >> input;
if (input == 'p' || input == 'P') {
cout << "\nEvent list:" ;
for (list<Event>::iterator it = event_list.begin(); it != event_list.end(); it++)
cout << ' ' << it->time << "(" << it->customer.id << ")";
cout << endl;
} else if (input == 'i' || input == 'I') {
cout << "Time: ";
cin >> input2;
}
}
}

```

```

list_insert(event_list, Event(input2, Customer(id++,0,DATA,input2), ARRIVAL));
} else if (input == 'r' || input == 'R') {
cout << "ID: ";
cin >> input3;
list_remove(event_list, input3);
}

cout << "COMMANDS:\n\tP = Print event list\n\tI = Insert event\n\tR = Remove event\n\tE
or Q = Exit test\n\n";

}

cout << "\n";
}

// Function that determines the size of a data package (actual function)
int data_package_size_test() {
long double seed = rand1();
if (seed < .3) return 64;
else if (seed < .3 + .1) return 512;
else if (seed < .3 + .1 + .3) return 1500;
else {
seed = (seed - .3 - .1 - .3) / (1 - .3 - .1 - .3);
return seed * (1500 - 64) + 64;
}
}

// Function that determines the number of voice packages (actual function)
int voice_package_number_test() {
long double seed = (long double) rand1();
if (seed == 1) seed = .999999; // seed cannot be 1 or there would be a log(0), which
is infinite,
// so... we make it a little smaller
return (int) ceil(log(1.0 - seed) / log(1 - 1.0L / 22.0));
}

// This function only exists to make sure that the percentages are correct
void sample_mean_test(int (*function)()) {
int current_sample;
int n64 = 0;
int n512 = 0;
int n1500 = 0;
for (int i = 0; i < BIGNUM; i++) {
current_sample = (*function)();
if (current_sample == 64) n64++;
else if (current_sample == 512) n512++;
else if (current_sample == 1500) n1500++;
}
}

```

```

}

cout << "\t\t64: " << (double)n64/BIGNUM * 100 << "%\n\t\t512: " <<
(double)n512/BIGNUM * 100 << "%\n\t\t1500: " << (double)n1500/BIGNUM * 100 << "%" << endl;

int main() {
unsigned int random;
cout << "Random Seed: ";
cin >> random;
srand(random);

cout << endl;

cout << "Data Package Size" << endl;
cout << "\tSample Mean: " << sample_mean(&data_package_size_test) << " bytes\n";
sample_mean_test(&data_package_size_test);

cout << endl;

cout << "Voice Package Quantity" << endl;
cout << "\tSample Mean: " << sample_mean(&voice_package_number_test) << endl;

cout << endl;

cout << "Exponential Silence Period" << endl;
cout << "\tSample Mean: " << sample_mean(&exponential,(double)1.0/650) << "ms" << endl;

cout << endl;

cout << "Event List test" << endl;
event_list_test();

cout << "Queue System test" << endl;
queue_test();

cout << endl;

cout << "Voice Package test" << endl;
cout << "\tDeterministic Silence Period: " << endl;
round_test(10,200,50,0.16,false,true,false);

cout << endl;

cout << "\tDeterministic Data Traffic: " << endl;
round_test(10,200,50,0.16,false,true,true);

```

}

5 Tabela com Resultados

Para avaliar os resultados do programa, foram feitos diversos testes com todas as utilizações possíveis, com semente = 5, período trasiente = 10000, quantidade de fregueses por rodada = 20000 e quantidade de rodadas = 100.

5.1 Casos Sem Preempção

Table 1: Utilização = 0.1

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	3.28613	3.30135	3.31657
E[W1]	0.40346	0.412317	0.421174
E[X1]	2.87787	2.88903	2.9002
E[Nq1]	0.0140137	0.0143494	0.0146851
E[T2]	0.551279	0.55372	0.556161
E[W2]	0.305625	0.30805	0.310474
E[Nq2]	0.20041	0.202393	0.204377
E[Delta]	15.9992	15.9996	15.9999
V[Delta]	1.81716	1.83174	1.84633

Table 2: Utilização = 0.2

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	3.79981	3.81545	3.83109
E[W1]	0.917996	0.929669	0.941342
E[X1]	2.87724	2.88579	2.89434
E[Nq1]	0.0636149	0.0645104	0.065406
E[T2]	0.836497	0.839937	0.843378
E[W2]	0.589992	0.593424	0.596855
E[Nq2]	0.389281	0.392708	0.396135
E[Delta]	15.9987	15.9992	15.9997
V[Delta]	3.27768	3.29921	3.32075

Conforme a utilização do pacote de dados vai aumentando, o tempo de espera deles também aumenta. Isso se deve ao fato de que a utilização é a probabilidade de ocupação, logo, quanto maior a probabilidade de estar ocupado, mais tempo vai levar para outros pacotes serem atendidos. Como a política implementada é sem preempção, o tempo gasto no servidor pelo pacote de dados não é alterado pela chegada de novos pacotes de voz, e se mantém estável nas utilizações.

Table 3: Utilização = 0.3

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	4.50143	4.52532	4.54922
E[W1]	1.62342	1.64345	1.66348
E[X1]	2.87458	2.88189	2.88919
E[Nq1]	0.169443	0.171798	0.174153
E[T2]	1.12855	1.13341	1.13828
E[W2]	0.881243	0.886066	0.890889
E[Nq2]	0.578653	0.583472	0.588291
E[Delta]	15.9994	15.9999	16.0005
V[Delta]	4.40301	4.4262	4.44939

Table 4: Utilização = 0.4

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	5.50881	5.54214	5.57547
E[W1]	2.62646	2.65677	2.68709
E[X1]	2.87904	2.88536	2.89168
E[Nq1]	0.364633	0.369381	0.374128
E[T2]	1.42024	1.42567	1.4311
E[W2]	1.17216	1.1775	1.18284
E[Nq2]	0.772626	0.777741	0.782856
E[Delta]	15.9992	15.9999	16.0005
V[Delta]	5.15603	5.17606	5.19609

Table 5: Utilização = 0.5

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	7.15999	7.21757	7.27515
E[W1]	4.27506	4.3306	4.38614
E[X1]	2.88141	2.88695	2.8925
E[Nq1]	0.740851	0.751248	0.761645
E[T2]	1.71066	1.71665	1.72263
E[W2]	1.46181	1.4677	1.47359
E[Nq2]	0.96308	0.97009	0.9771
E[Delta]	15.9994	16.0001	16.0008
V[Delta]	5.55348	5.57499	5.59651

Além disso, outras métricas, como por exemplo, a variância dos intervalos de pacotes de voz também é afetada pelo aumento da utilização, porque apesar de ter prioridade, como não é preemptivo, tem que esperar o pacote de voz acabar de ser servido, como a probabilidade de ocupação é maior do que nos outros

Table 6: Utilização = 0.6

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	10.2864	10.4224	10.5584
E[W1]	7.40505	7.53925	7.67345
E[X1]	2.87844	2.88315	2.88787
E[Nq1]	1.54111	1.57119	1.60127
E[T2]	2.00461	2.01249	2.02038
E[W2]	1.75503	1.76277	1.77052
E[Nq2]	1.15194	1.16094	1.16995
E[Delta]	15.9991	15.9998	16.0005
V[Delta]	5.70553	5.72452	5.74352

Table 7: Utilização = 0.7

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	18.3046	18.7447	19.1849
E[W1]	15.4245	15.8621	16.2996
E[X1]	2.87713	2.88262	2.88811
E[Nq1]	3.7531	3.86408	3.97507
E[T2]	2.30279	2.31183	2.32087
E[W2]	2.05259	2.06142	2.07026
E[Nq2]	1.35234	1.36326	1.37418
E[Delta]	15.9996	16.0003	16.0009
V[Delta]	5.62715	5.6439	5.66065

casos, no total os intervalos vão ter uma discrepância maior (até chegar a um ponto de equilíbrio) em relação a média, aumentando a variância. Basicamente, a maioria das métricas aumenta com a utilização, o que já era esperado.

5.2 Casos Com Preempção

No caso da preempção, temos com o aumento da utilização do pacote de dados, assim como no caso anterior, o aumento da quantidade de tempo de espera do pacote de dados na fila, mas nesse caso, a grande quantidade se deve, também, ao fato de que com preempção, a chegada de um pacote de voz interrompe o serviço do pacote de dados, fazendo com que ele espere mais tempo, até porque voz também tem prioridade, e acaba complicando ainda mais a situação para os dados. De qualquer forma, é possível verificar que nos casos sem preempção, o aumento da utilização impactava bem mais o tempo de espera dos dados, porque em casos preemptivos, como eles já tem um tempo de espera alto por causa das vozes, acabam entrando em um certo equilíbrio e não variando tanto depois (como no caso de utilização = 0.4, utilização = 0.5 e 0.6)

Outra observação que é importante de ser feita é entre os casos que tem

Table 8: Utilização = 0.1

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	333448	375248	417048
E[W1]	333418	375216	417014
E[X1]	29.2021	31.8552	34.5083
E[Nq1]	11534.5	12689.6	13844.8
E[T2]	0.267228	0.267683	0.268138
E[W2]	0.0224193	0.0228676	0.0233159
E[Nq2]	0.0147135	0.0150358	0.0153582
E[Delta]	15.9995	15.9995	15.9996
V[Delta]	0.000375701	0.000384029	0.000392357

Table 9: Utilização = 0.2

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	796126	930759	1.06539e+06
E[W1]	796096	930724	1.06535e+06
E[X1]	30.4079	35.1454	39.8829
E[Nq1]	57410	63362.3	69314.7
E[T2]	0.267328	0.267786	0.268244
E[W2]	0.022511	0.0229534	0.0233958
E[Nq2]	0.0148742	0.0152111	0.015548
E[Delta]	15.9995	15.9996	15.9996
V[Delta]	0.000367646	0.000376734	0.000385822

Table 10: Utilização = 0.3

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	883043	1.03985e+06	1.19665e+06
E[W1]	883014	1.03981e+06	1.19662e+06
E[X1]	28.4148	33.2184	38.0219
E[Nq1]	95285.9	105218	115149
E[T2]	0.266737	0.267143	0.267549
E[W2]	0.0219236	0.0223231	0.0227227
E[Nq2]	0.0144181	0.0147074	0.0149966
E[Delta]	15.9995	15.9996	15.9996
V[Delta]	0.000368844	0.000376995	0.000385146

preempção e os que não tem. Avaliando ambos, é possível ver que nos casos em que ocorre preempção, o tempo total (considerando espera e execução) dos pacotes de voz é bem baixo devido a interrupção no serviço dos pacotes de dados e serviço quase imediato na chegada a fila, com isso, fazendo com

Table 11: Utilização = 0.4

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	943156	1.11451e+06	1.28586e+06
E[W1]	943128	1.11447e+06	1.28582e+06
E[X1]	27.6215	32.4556	37.2896
E[Nq1]	129744	143262	156780
E[T2]	0.267377	0.267899	0.268421
E[W2]	0.0225418	0.0230503	0.0235589
E[Nq2]	0.014879	0.0152392	0.0155994
E[Delta]	15.9995	15.9996	15.9996
V[Delta]	0.000370293	0.000379382	0.000388471

Table 12: Utilização = 0.5

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	921440	1.09225e+06	1.26307e+06
E[W1]	921412	1.09222e+06	1.26303e+06
E[X1]	27.5808	32.5425	37.5042
E[Nq1]	162181	179101	196021
E[T2]	0.267051	0.267558	0.268066
E[W2]	0.0222155	0.0227109	0.0232063
E[Nq2]	0.0146636	0.0150353	0.0154069
E[Delta]	15.9996	15.9996	15.9997
V[Delta]	0.000369059	0.000377618	0.000386178

Table 13: Utilização = 0.6

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	964593	1.14462e+06	1.32464e+06
E[W1]	964567	1.14459e+06	1.32461e+06
E[X1]	25.9595	30.7117	35.4638
E[Nq1]	192914	213001	233088
E[T2]	0.266772	0.267236	0.267699
E[W2]	0.0219329	0.0223866	0.0228403
E[Nq2]	0.0144284	0.0147606	0.0150928
E[Delta]	15.9996	15.9996	15.9997
V[Delta]	0.000375185	0.000384943	0.000394701

que a variância entre os intervalos também se torne bem pequena, porque quase todos possuem o mesmo valor, já que são atendidos rapidamente. Enquanto isso, os pacotes de dados sofrem de valores bem maiores com relação ao tempo em que esperam pelos motivos citados acima, de que além de esperar para ser

Table 14: Utilização = 0.7

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	887680	1.05576e+06	1.22384e+06
E[W1]	887652	1.05573e+06	1.22381e+06
E[X1]	28.2652	33.5252	38.7851
E[Nq1]	220703	243765	266828
E[T2]	0.267444	0.267997	0.268549
E[W2]	0.022581	0.023126	0.023671
E[Nq2]	0.0149142	0.0153139	0.0157136
E[Delta]	15.9996	15.9997	15.9997
V[Delta]	0.000373411	0.000384426	0.000395441

servido só quando a fila de voz estiver vazia, também são interrompidos e tem que esperar todas as vozes serem servidas para poderem retornar.

5.3 Semente diferente

O teste foi realizado para os mesmos valores acima, com serviço sem preempção e semente = 7

Table 15: Utilização = 0.5

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	7.17929	7.24087	7.30246
E[W1]	4.30054	4.35973	4.41892
E[X1]	2.87508	2.88113	2.88718
E[Nq1]	0.74736	0.758678	0.769995
E[T2]	1.71011	1.71728	1.72445
E[W2]	1.4613	1.46836	1.47542
E[Nq2]	0.961235	0.968993	0.97675
E[Delta]	15.9995	16.0002	16.0009
V[Delta]	5.55801	5.57987	5.60172

e com serviço com preempção e semente = 7

Como podemos verificar nessa tabela, o valor encontrado para os resultados estão muito perto do valor encontrado para a semente = 5. Além disso, todos os intervalos de confiança têm largura menor do que 10 porcento da média, como requisitado.

Table 16: Utilização = 0.5

Métricas	Limite Inferior	Valor médio	Limite Superior
E[T1]	975177	1.15542e+06	1.33567e+06
E[W1]	975148	1.15539e+06	1.33563e+06
E[X1]	28.3795	33.4885	38.5975
E[Nq1]	163869	180979	198089
E[T2]	0.266703	0.267212	0.267721
E[W2]	0.0218734	0.02237	0.0228667
E[Nq2]	0.0144148	0.014792	0.0151692
E[Delta]	15.9996	15.9996	15.9997
V[Delta]	0.000365184	0.000376415	0.000387646

6 Conclusões

O objetivo do trabalho, de trazer uma visão prática para a teoria aprendida durante o curso foi alcançado, visto que muito foi aprendido durante a implementação do mesmo, e até ficou melhor de ver os resultados de como cada situação pode e deve modificar os resultados finais da simulação.

No começo da implementação, o mais complicado era entender como juntar cada um dos dados que nos foi passado, e por isso a melhor ideia foi começar implementando as partes, para depois entrelaçar no todo. Inclusive, a apostila se mostrou muito útil para compreender a estrutura de um simulador e como deveriam ser implementadas as filas de eventos.

Com o passar da implementação, tudo começou a fazer sentido e as partes foram se encaixando, fazendo com que se tornasse mais fluida a programação do simulador.

Os testes também estavam sendo implementados junto com o programa, pois era a forma mais prática de termos certeza de que todas as funções estavam trabalhando corretamente, e com isso, a importância dele no trabalho é totalmente compreendida, pois serve para nós também termos uma noção se está tudo correndo bem.

Um dos piores problemas encontrados se deve a linguagem, pois como foi explicado acima, apesar de ser uma ótima linguagem, não existem bibliotecas gráficas de fácil acesso para ela, e por isso, foi necessária a escrita em um arquivo externo dos dados para serem representados e depois a implementação de um programa em Python para a plotagem desse gráfico, como nas imagens mostradas na estimativa da fase transiente. Isso acabou gerando ainda mais trabalho, na intenção de entregar um programa mais completo.