

MAB-515: Avaliação e Desempenho

Trabalho Final

Alunos: Bruno Behnken, Gabriel Bevilaqua e Pedro Coutinho

Neste trabalho, o aluno Bruno fez o gerador de amostras exponenciais (classe ExpGenerator), o agendador de eventos (classe Scheduler), o cliente (classe Client), os métodos simulate_FCFS (classe SimulatorFCFS) e simulate_LCFS (classe SimulatorLCFS), e os métodos run_FCFS e run_LCFS da classe Master.

O aluno Gabriel fez a classe Web.py, os templates home.html e simulate.html (incluindo o *Bootstrap* e o plot de gráficos utilizando o *ChartJS*), bem como fez a análise dos gráficos para determinar o valor de Kmin.

O aluno Pedro fez a análise da fase transiente e o método transient_phase (classes SimulatorFCFS e SimulatorLCFS).

Os três componentes do grupo trabalharam juntos na programação do método webmain (classe Master) e na programação da classe Statistics.

Este relatório é referente ao release v0.2-beta do simulador.

Bruno

Gabriel

Pedro

1) Introdução

A linguagem escolhida para fazer o simulador foi o Python, pois conta com facilidades na manipulação de estruturas de dados. Há diversas operações pré-programadas para estruturas como listas, por exemplo, que foram amplamente utilizadas no trabalho. Essa característica da linguagem permitiu aos membros focar o esforço de programação no trabalho em si, ao invés de se preocupar com a complexidade envolvida na programação de operações como inserção e remoção de elementos em listas, etc.

Também foi utilizada a facilidade da geração de números aleatórios da linguagem, que consiste em uma amostra da distribuição Uniforme no intervalo [0,1]. Esta facilidade foi utilizada na classe ExpGenerator, que foi programada pelos membros do grupo para gerar amostras de distribuições exponenciais. Esta classe recebe o parâmetro lambda da distribuição exponencial e possui métodos para gerar uma amostra e uma lista de amostras utilizando o lambda fornecido, bem como um método para gerar uma amostra utilizando lambda=1 (independente do lambda fornecido anteriormente), que foi criado especialmente para gerar os tempos de serviço, uma vez que o enunciado informa que o tempo médio de serviço será igual a 1 segundo. Esta classe também recebe uma semente para alimentar o gerador da distribuição U(0,1), que por padrão é o valor Null, o que deixa a critério do gerador decidir a própria semente. Entretanto, caso seja necessário ou desejado, é possível alterar esta semente na inicialização do simulador.

A classe ExpGenerator é utilizada sempre pela classe Scheduler. Esta foi programada para ser um agendador de eventos, que mantém o controle sobre a lista de próximos eventos

do simulador. Esta classe possui apenas três métodos públicos: um para agendar um evento de chegada, um para agendar um evento de partida e um para obter o próximo evento a ser processado pelo simulador. Cada elemento da lista de eventos possui duas informações: um caractere que identifica o tipo do evento (os eventos podem ser do tipo chegada, identificados pelo caractere ‘a’; ou do tipo partida, identificados pelo caractere ‘d’) e um objeto da classe Client, que representa o cliente associado àquele evento. A lista de eventos sempre está ordenada em ordem crescente de tempo.

A classe Client foi programada como uma *struct* para que suas instâncias representem um cliente. Ela guarda quatro informações: os tempos de chegada, espera, serviço e partida. Os tempos de chegada e serviço devem ser passados como parâmetro no momento em que o objeto desta classe é instanciado, e não podem ser alterados posteriormente. Os tempos de espera e partida são alterados pelo servidor conforme o cliente vai sendo atendido.

A classe Scheduler é utilizada pelas classes SimulatorFCFS e SimulatorLCFS. Ambas as classes possuem apenas dois métodos públicos, um para realizar a simulação de acordo com a disciplina desejada e um para executar a fase transiente do programa (que será abordada em seção específica mais adiante). Os dois métodos possuem uma mecânica de funcionamento com a classe Scheduler, de modo que as classes Simulator recebem o primeiro evento, que é uma chegada, e solicitam o agendamento do evento da partida do cliente que está sendo processado, bem como o agendamento do próximo evento de chegada. A partir daí, o Simulator solicita o próximo evento da lista, e essa mecânica se repete enquanto durar a fase transiente ou a rodada de simulação, dependendo do método que foi chamado. O término da rodada de simulação acontece quando o número de clientes servidos for igual ao tamanho da rodada.

As classes Simulator também possuem instâncias da classe Statistics, que foi desenhada para realizar os cálculos de estatística. A classe Simulator faz a coleta das estatísticas e as passa como parâmetro para os métodos de cálculo de média, variância e intervalo de confiança da classe Statistics. Os resultados desses cálculos são armazenados em variáveis do tipo lista da classe Simulator que serão retornados para quem chamou o método.

As estatísticas são coletadas pelas classes Simulator da seguinte forma: quando um evento de chegada de cliente é processado, verifica-se se o servidor está ocioso. Se estiver, o cliente é atendido e a lista de estatísticas de tempo de espera é incrementada com um “zero”, pois o cliente não precisou esperar pelo atendimento. Se o servidor estiver ocupado, então a estatística a ser coletada será a do número de clientes na fila. Para isto, é utilizada uma variável delta_time que armazena a diferença entre o instante em que houve a última alteração no número de clientes na fila e o instante atual, onde acontece uma nova alteração. Esta variável é multiplicada pela quantidade de clientes na fila neste espaço de tempo, de modo que o resultado é a área sob o gráfico de tempo x clientes na fila de espera. Esta área é adicionada à lista de estatísticas do número de clientes na fila. Caso o evento processado seja uma partida, então é verificado se a fila possui algum cliente aguardando para ser atendido. Se possuir, esse cliente é chamado para o serviço, e a estatística do número de clientes na fila é coletada da mesma forma que foi descrita anteriormente, bem como é coletada a estatística do tempo de espera do cliente, calculando a diferença entre o tempo no qual ele chegou e o tempo

no qual ele foi atendido. Esta estatística é adicionada à lista de estatísticas de tempo de espera.

A classe Statistics realiza o cálculo dos estimadores fazendo uso de 8 métodos distintos. Ela possui um método para calcular a média de determinada lista de medidas e um para calcular a variância. Também possui um método para calcular a média e variância acumuladas e um outro específico para o cálculo da média acumulada do tempo de espera em fila. Para esses métodos a classe guarda os valores da soma das médias, do número de amostras das médias, da soma e número de amostras das variâncias, da soma dos quadrados das variâncias, da soma das médias dos tempos de espera e do número de amostras das médias dos tempos de espera até o momento atual. Esta classe também possui dois métodos para a estimativa dos intervalos de confiança, uma para o intervalo da média pela T-Student e um para o intervalo da variância pela Chi-Quadrado.

A chamada da classe Simulator é sempre realizada pela classe Master. Esta classe possui dois métodos públicos que instanciam o Simulator para as disciplinas FCFS e LCFS e realizam as 3200 rodadas de simulação, armazenando os resultados dos cálculos de estatística em listas para serem plotados em gráficos posteriormente. Estes métodos também são responsáveis por chamar o método para rodar a fase transiente. Além disso, a classe Master possui um método main que recebe os dados de input, chama o simulador da maneira adequada e prepara os dados para serem plotados na interface gráfica.

Finalmente, o método main da classe Master é chamado pelo arquivo Web, que é responsável por integrar o *front-end* (interface gráfica) com o *back-end* (simulador). Para esta integração, utilizamos o framework Flask, que roda em um servidor (que pode ser local) e renderiza os dois templates HTML que constituem a interface gráfica. Estes templates utilizam Bootstrap (HTML5) para possibilitar ao usuário fazer a seleção dos dados de entrada do simulador e utilizam a biblioteca ChartJS (javascript) para plotar os gráficos.

Máquina utilizada para rodar as simulações: placa mãe Asus H81M-E, processador Intel Quad Core i5-4670 CPU @ 3.40GHz, memória RAM 8GB DDR3 1600 MHz. Sistema Linux Mint 19 (Codinome Tara), 64 bits.

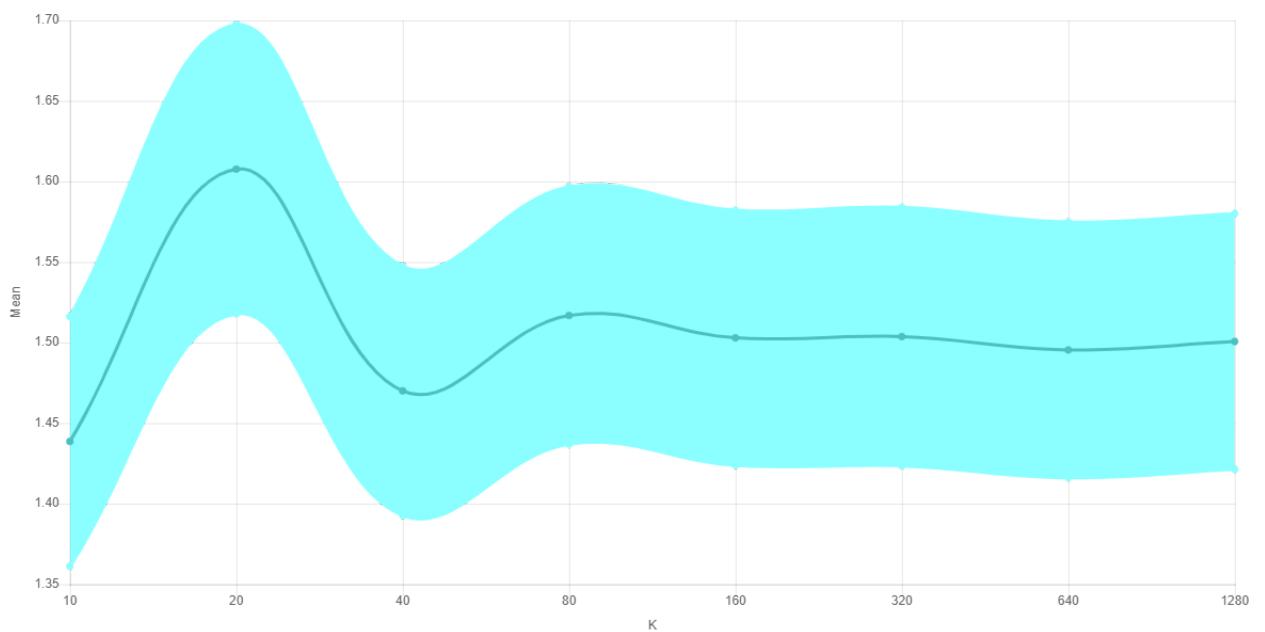
Disciplina FCFS.	Tempo de execução
Utilização = 0.2	20.2s
Utilização = 0.4	21.6s
Utilização = 0.6	22.6s
Utilização = 0.8	23.0s
Utilização = 0.9	25.2s

Disciplina LCFS.	Tempo de execução
------------------	-------------------

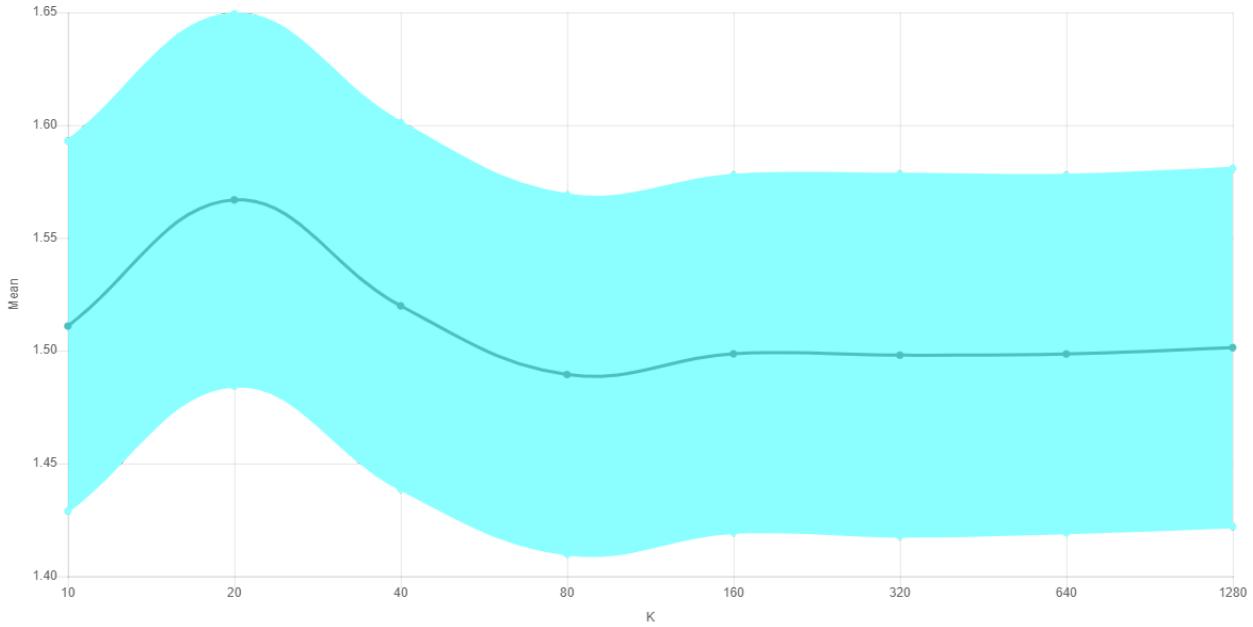
Utilização = 0.2	20.3s
Utilização = 0.4	21.2s
Utilização = 0.6	21.4s
Utilização = 0.8	23.3s
Utilização = 0.9	27.6s

A definição do kmin (número de coletas por rodada do método batch) para estimar a métrica da média do tempo de espera em fila foi feita rodando o simulador para distintos valores de k e analisando a média das rodadas do tempo de espera em fila. Após cada simulação o k é dobrado. O kmin foi definido para o valor 1280, onde a convergência da média pode ser claramente observada.

FCFS



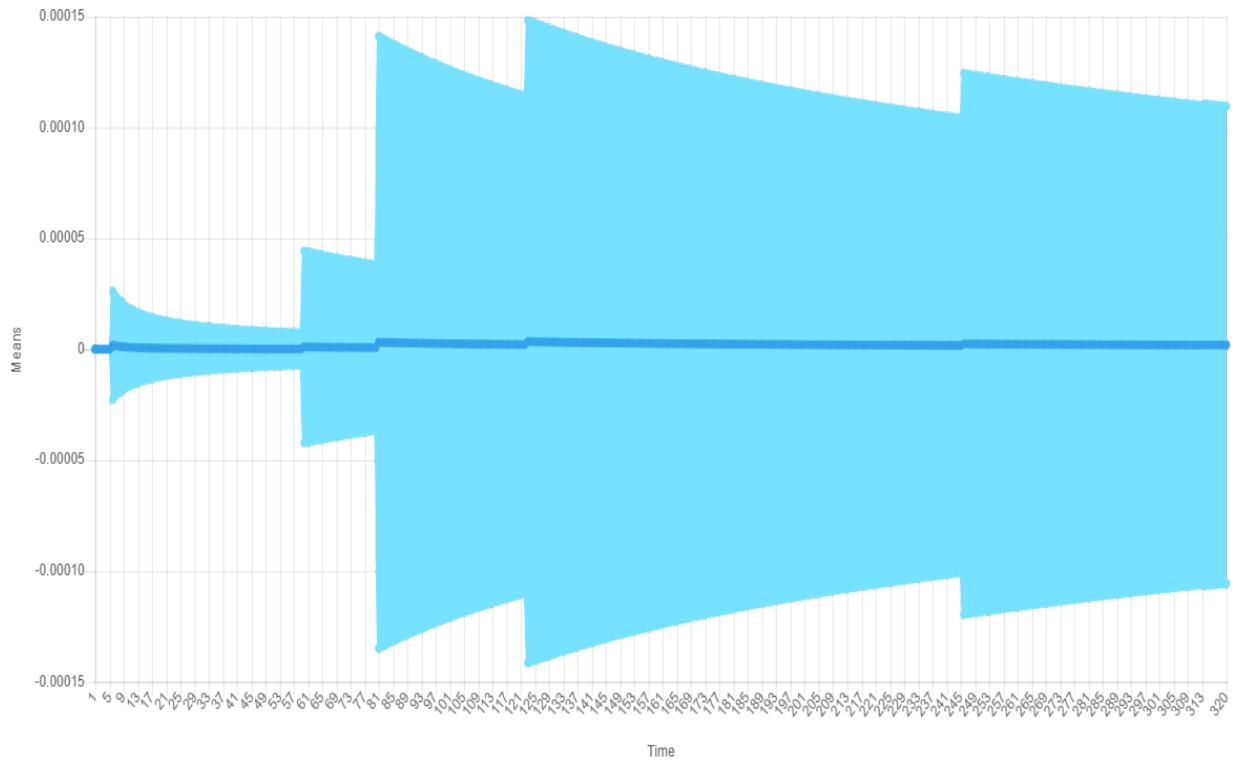
LCFS



2) Testes de Corretude

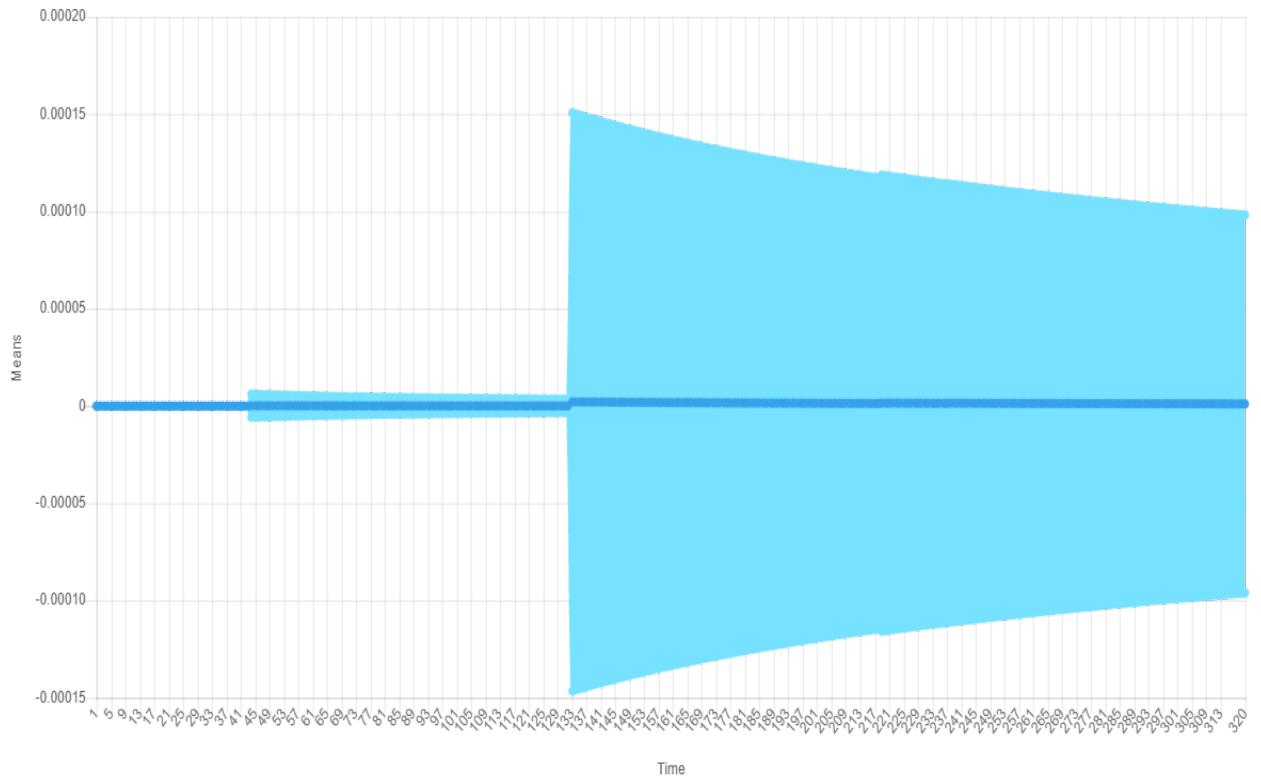
Para testar a corretude do nosso simulador o rodamos com a taxa de chegada de clientes muito baixa, pois nessa situação as estatísticas são conhecidas. Para uma taxa que tende a 0 o tempo de espera em fila tende a ser nulo e o tempo total gasto na fila tende a ser o tempo de serviço e foi isso que encontramos quando fizemos $\lambda = 0.000001$. O resultado pode ser observado nos gráficos abaixo:

FCFS



A média das rodadas dos tempos de espera converge para muito próximo de 0, paralelamente o tempo total de um cliente típico no sistema pode ser considerado como sendo em sua totalidade o tempo de serviço.

LCFS



3) Estimativa de Fase Transiente

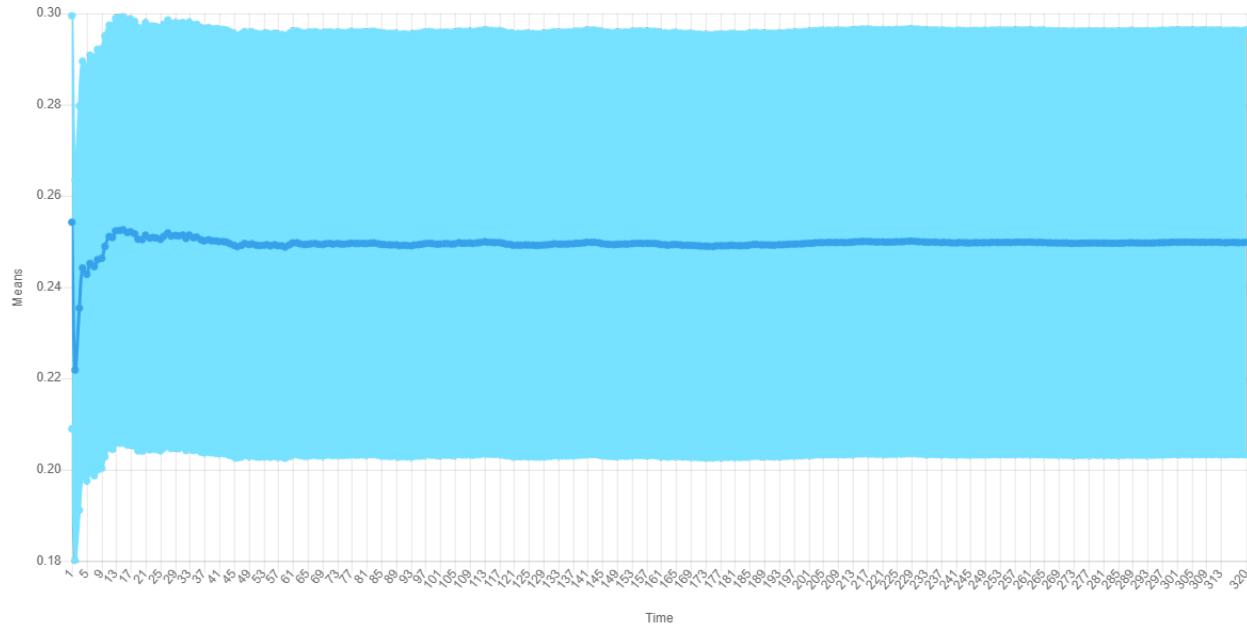
Para estimar a duração da fase transiente, o simulador realiza 10 rodadas de tamanho 100 e, para cada rodada, calcula a média dos tempos de espera ($E[W]$). Ao obter as 10 médias, o simulador calcula a variância dessa amostra de médias, e repete todo o procedimento. De posse das duas variâncias, é calculada a diferença entre elas, e caso essa diferença seja menor do que um limite pré-estabelecido, um contador é incrementado. Uma vez que o contador atinja um limite pré-estabelecido, a fase transiente é considerada encerrada. Para as utilizações 0.2, 0.4 e 0.6, o limite para a diferença estabelecido foi 10^{-4} e o limite para o contador foi estabelecido em 30. Para as utilizações 0.8 e 0.9, o limite para a diferença estabelecido foi 10^{-1} e o limite para o contador foi estabelecido em 150. Não há diferenças na estimativa da fase transiente entre as duas disciplinas do trabalho.

4) Resultados e Comentários

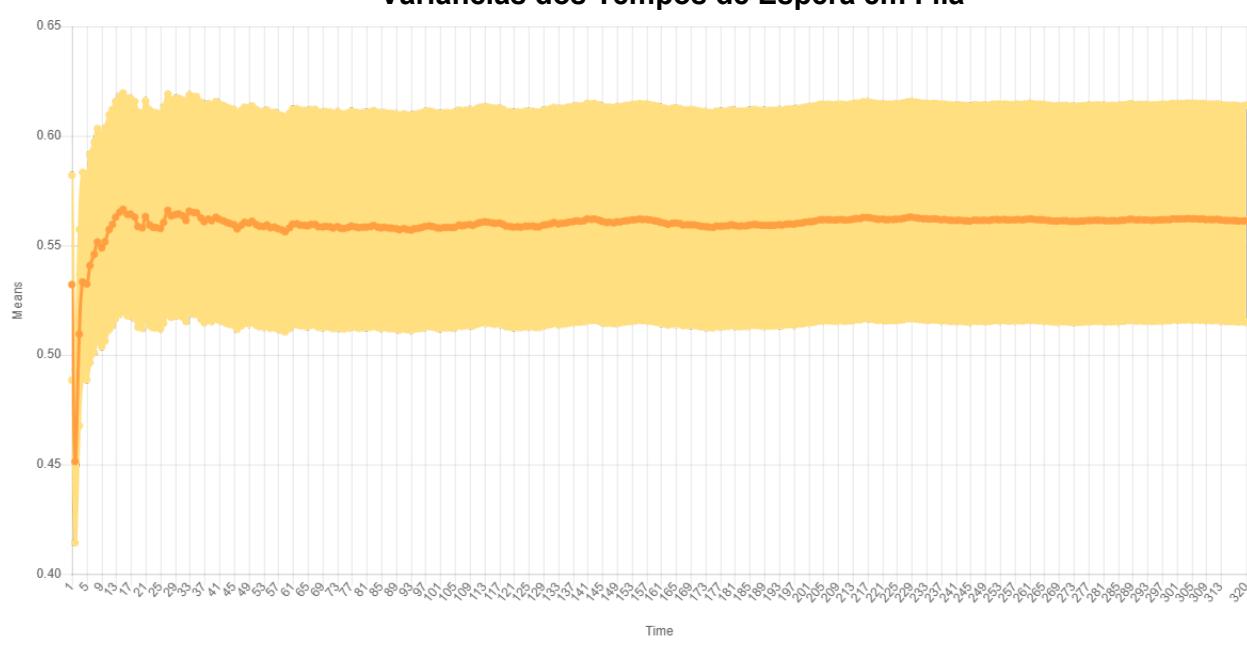
FCFS

Utilização: 0.2

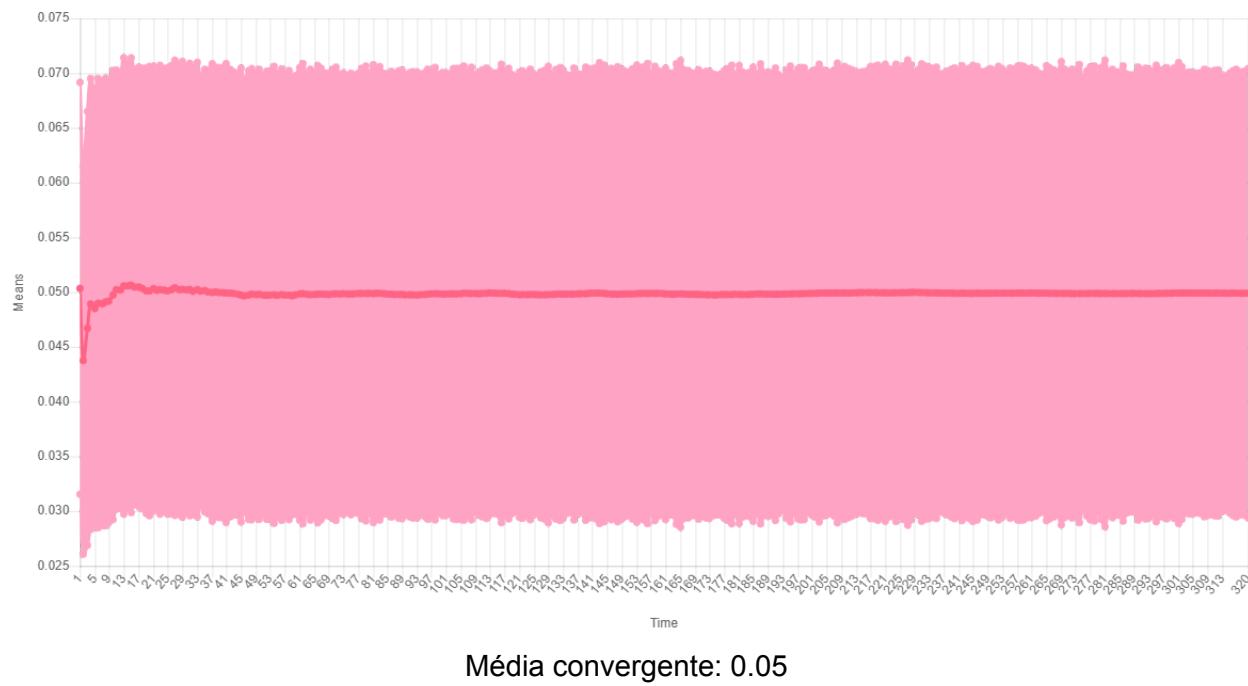
Médias dos Tempos de Espera em Fila



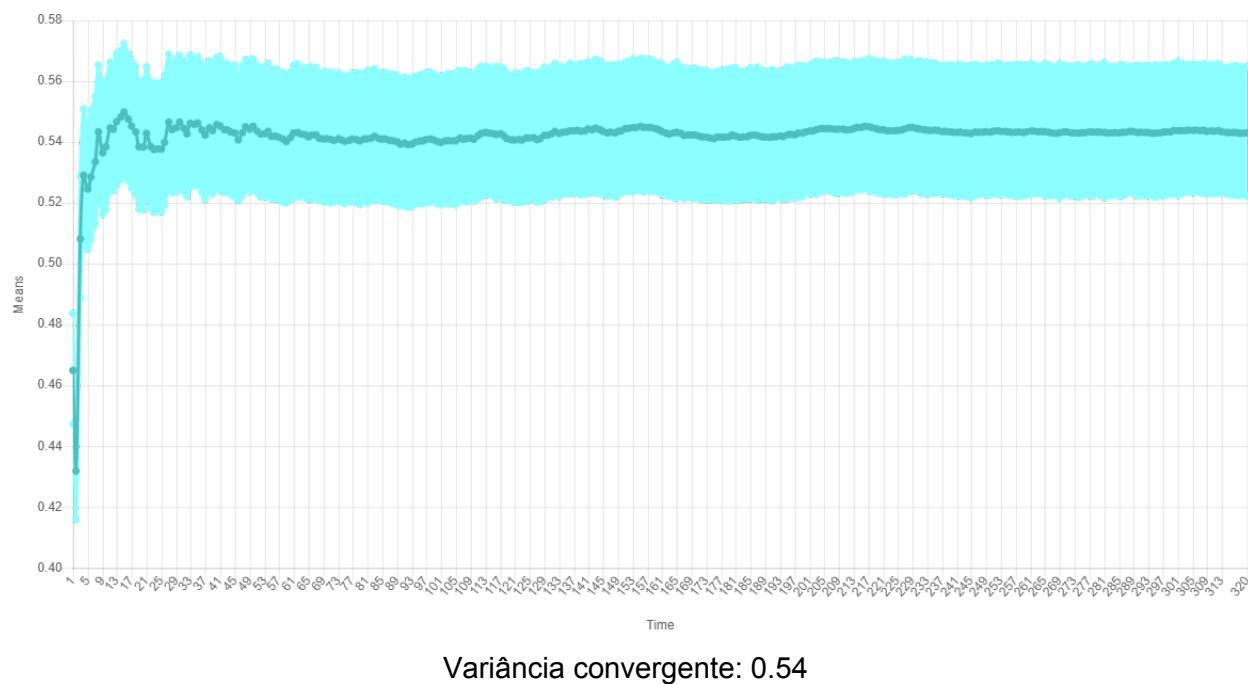
Variâncias dos Tempos de Espera em Fila



Médias do Número de Pessoas na Fila de Espera

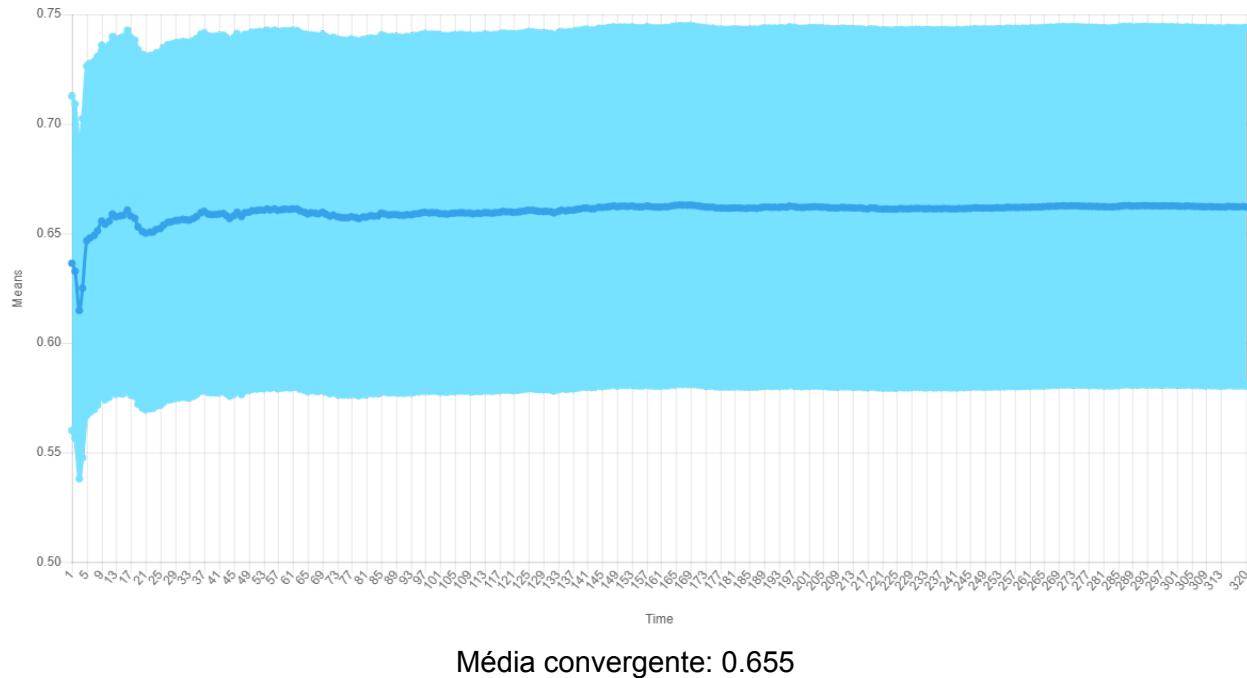


Variâncias do Número de Pessoas na Fila de Espera

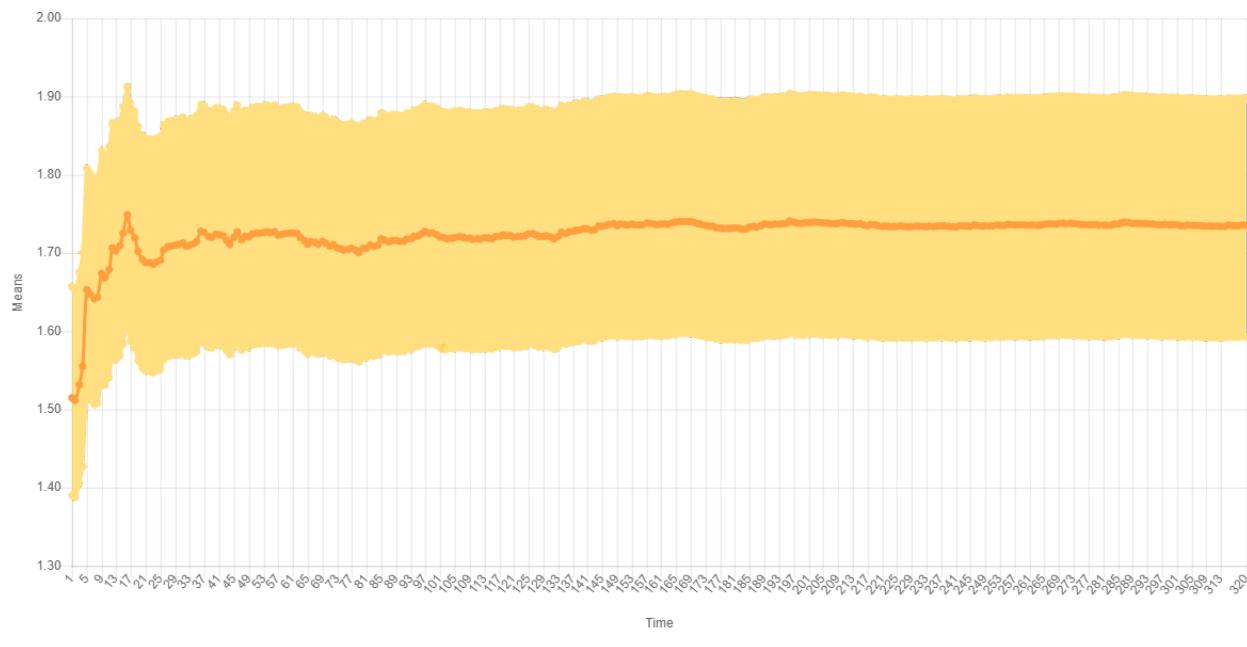


Utilização: 0.4

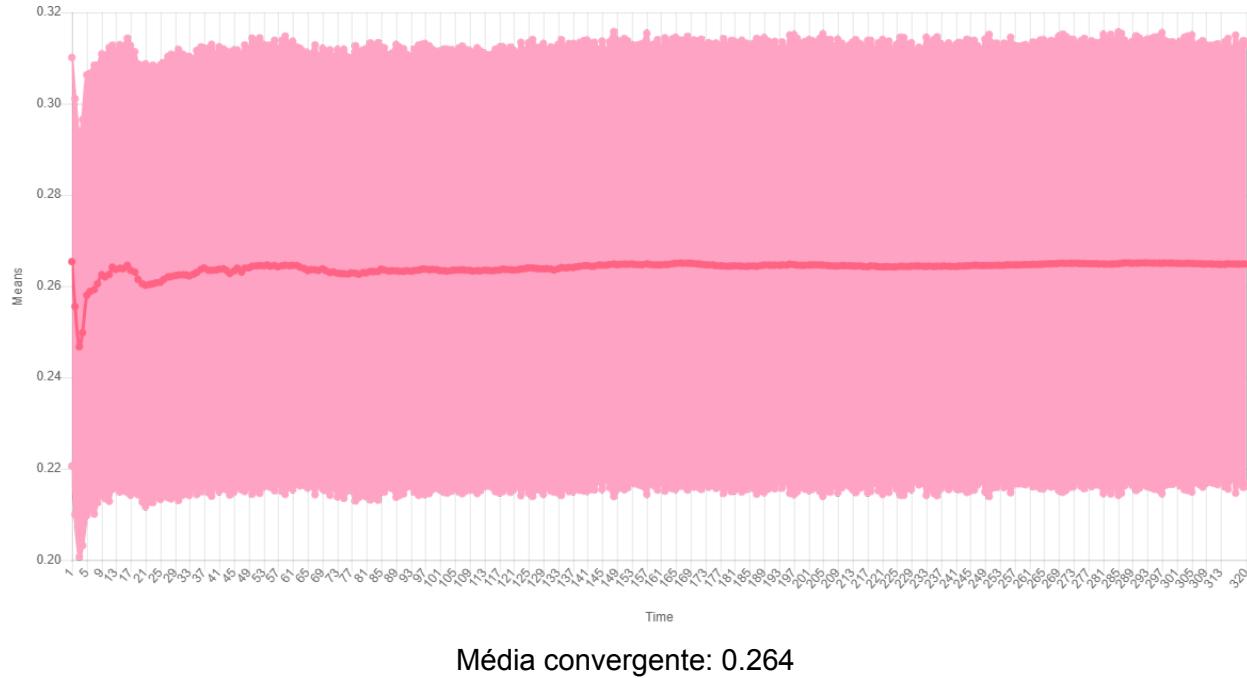
Médias dos Tempos de Espera em Fila



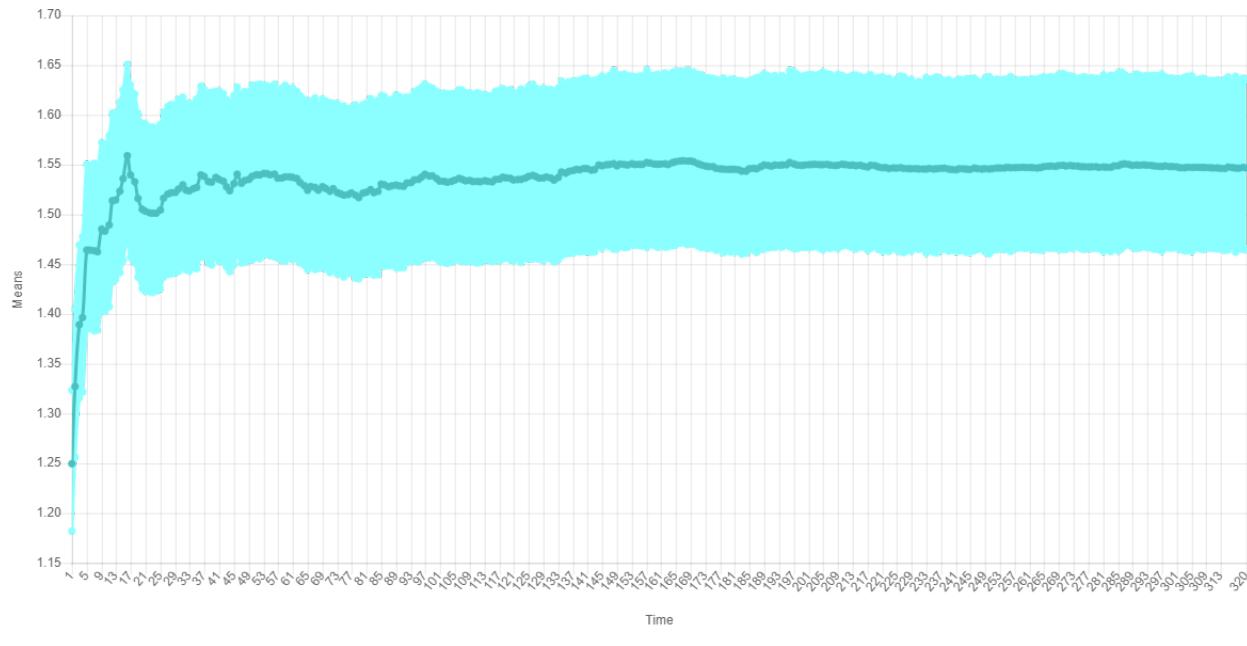
Variâncias dos Tempos de Espera em Fila



Médias do Número de Pessoas na Fila de Espera

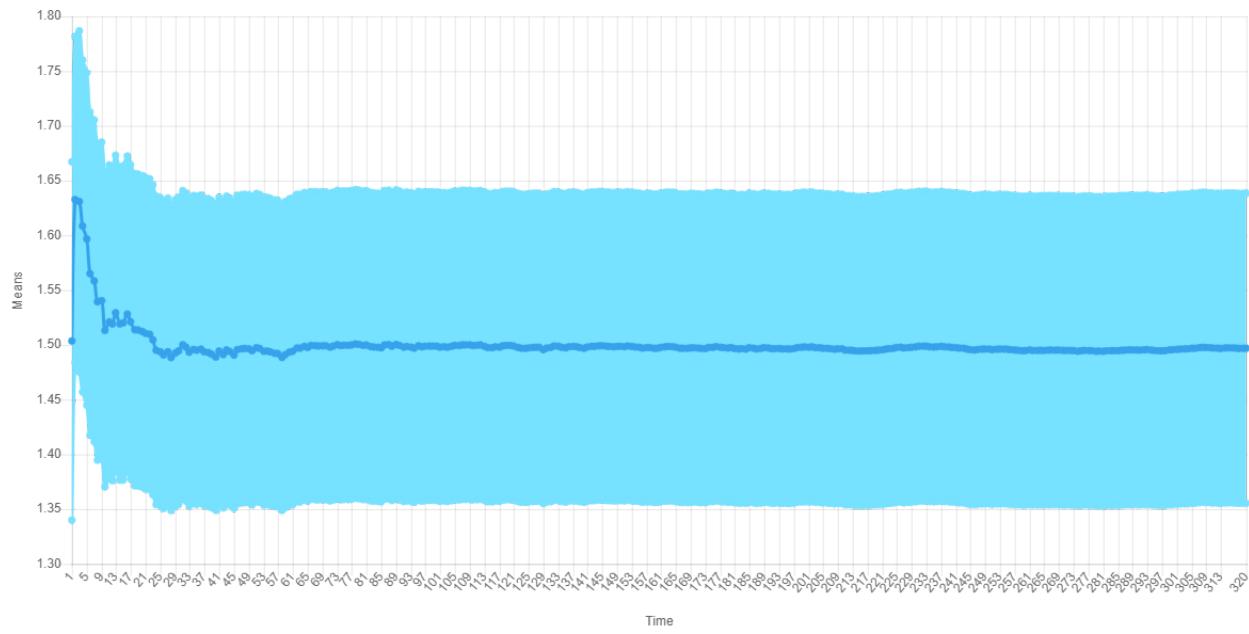


Variâncias do Número de Pessoas na Fila de Espera

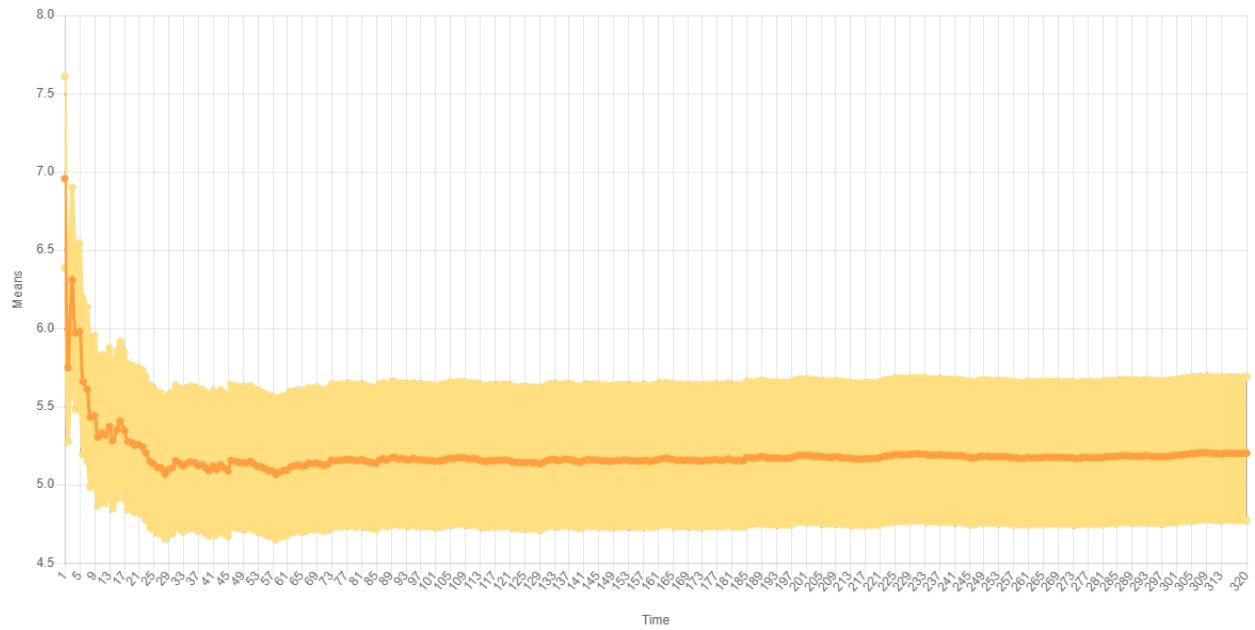


Utilização: 0.6

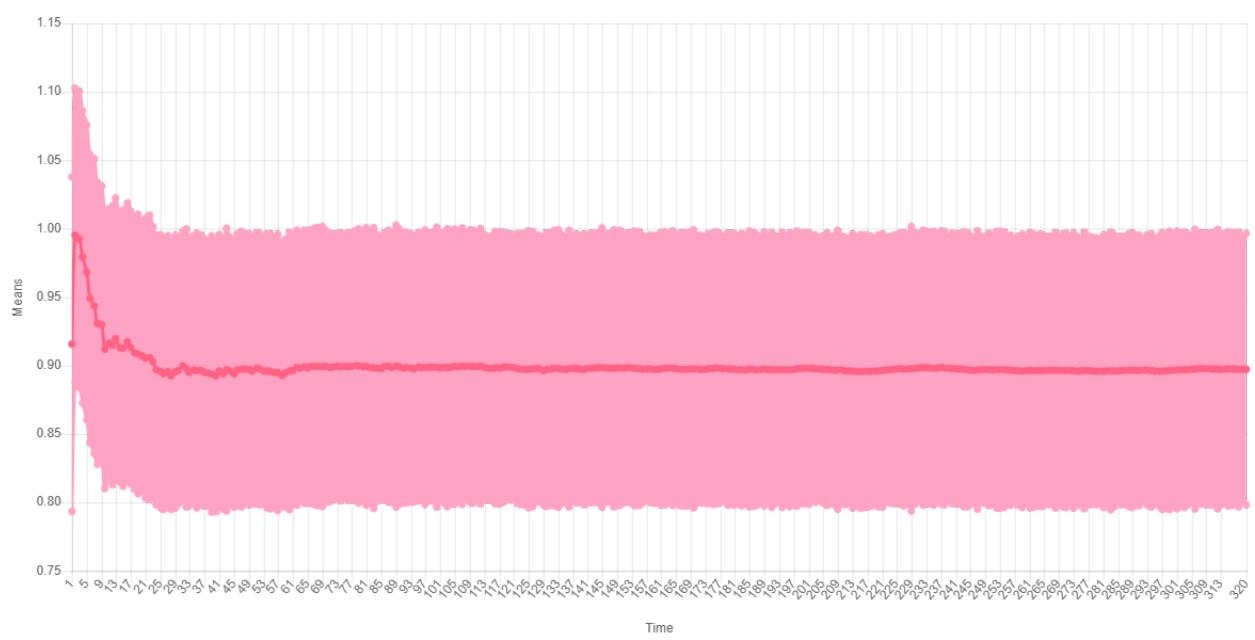
Médias dos Tempos de Espera em Fila



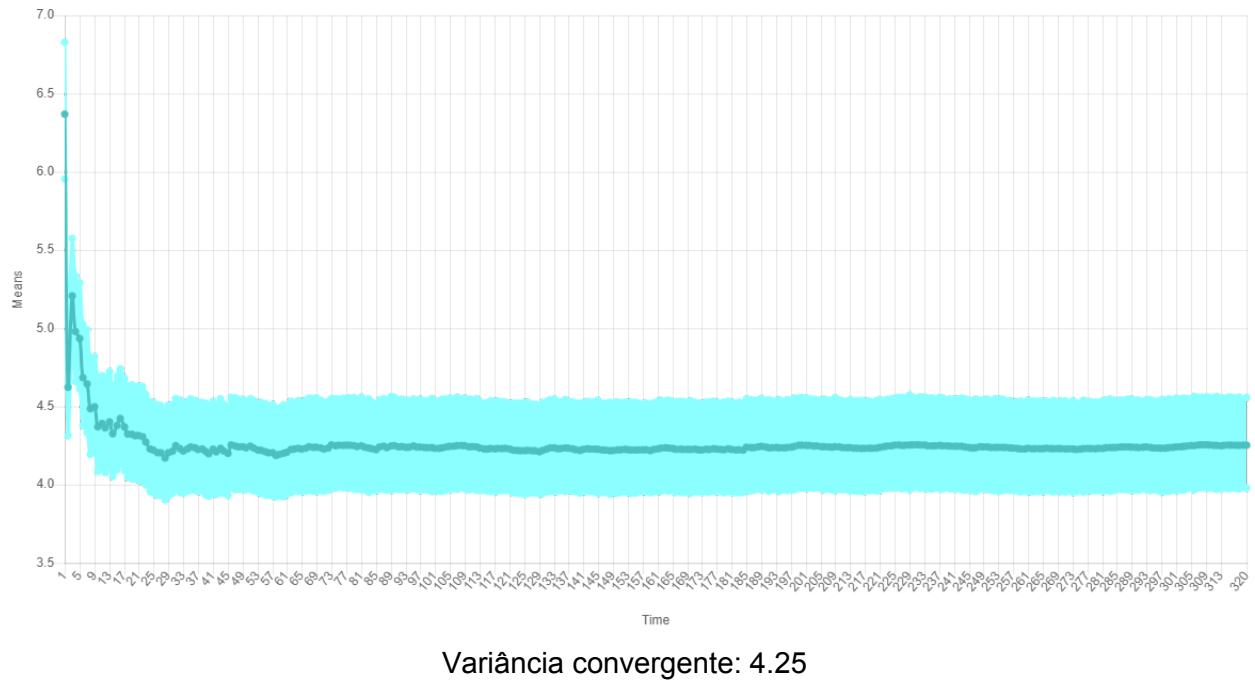
Variâncias dos Tempos de Espera em Fila



Médias do Número de Pessoas na Fila de Espera

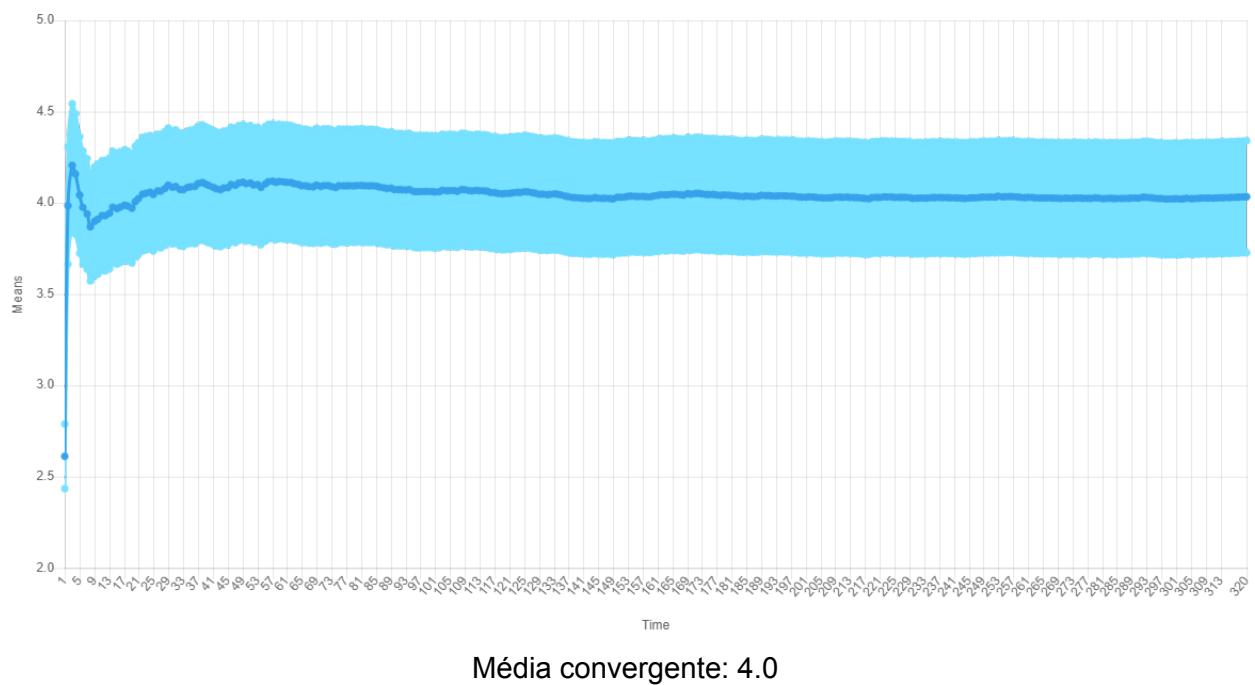


Variâncias do Número de Pessoas na Fila de Espera

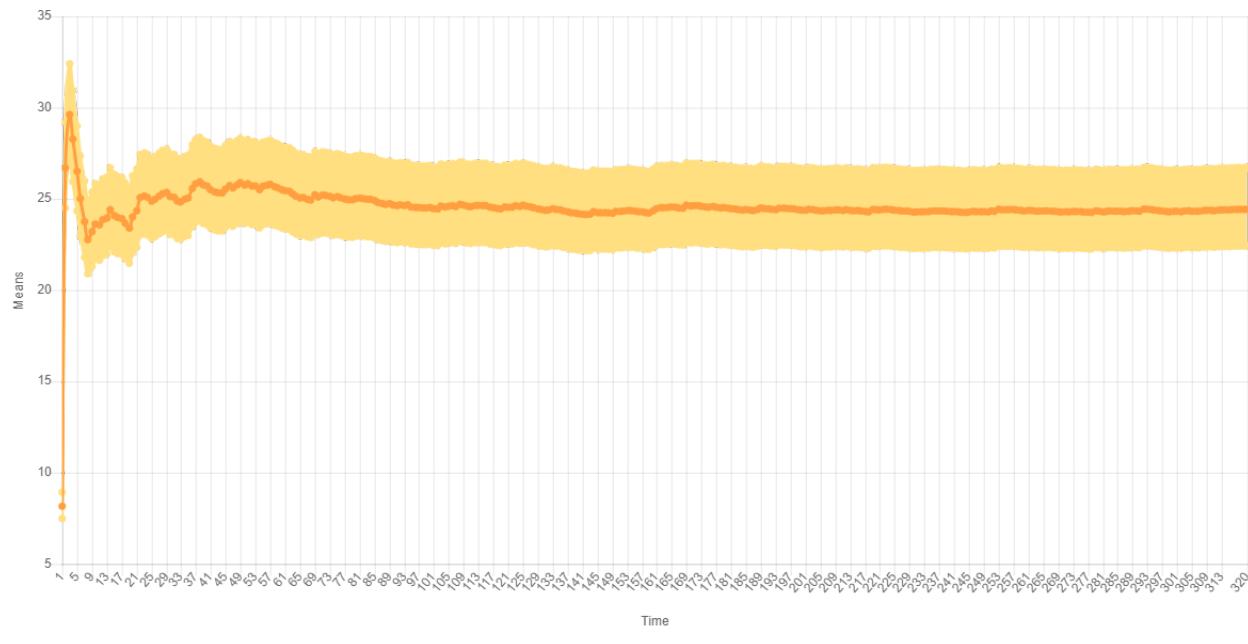


Utilização: 0.8

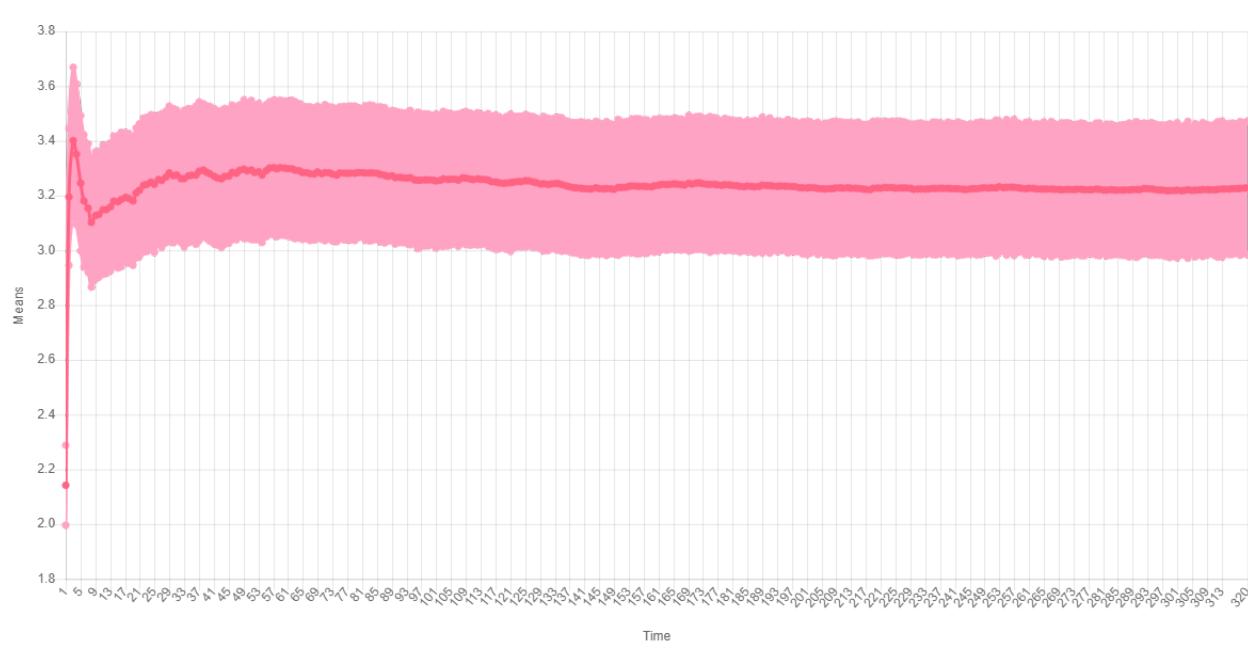
Médias dos Tempos de Espera em Fila



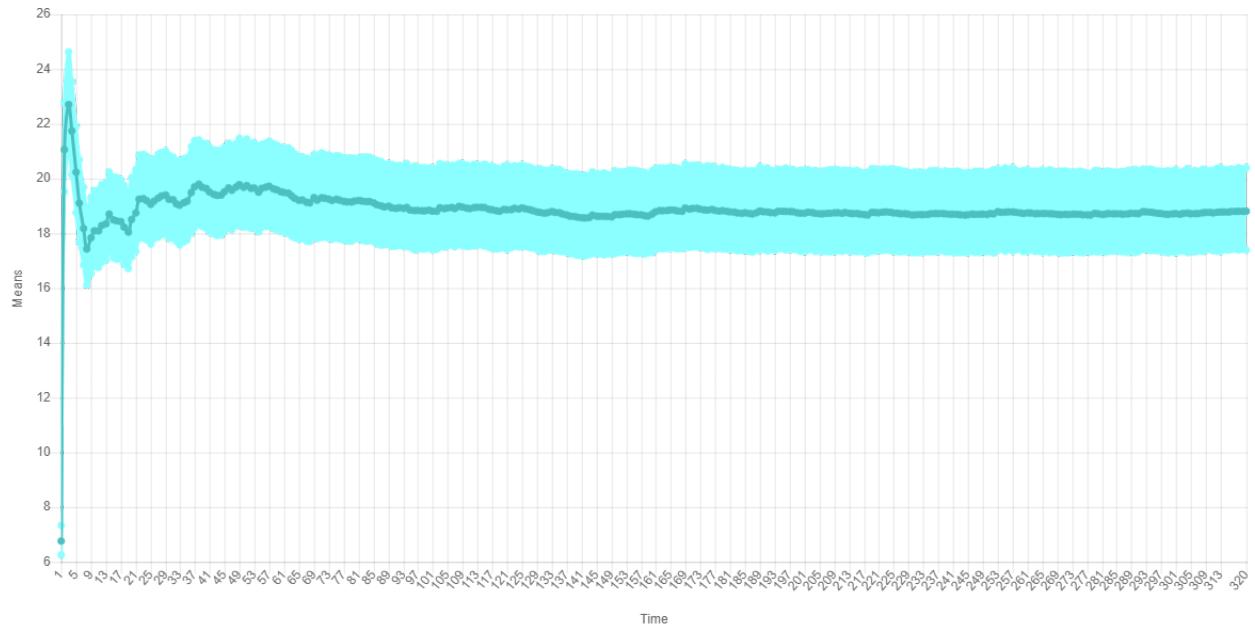
Variâncias dos Tempos de Espera em Fila



Médias do Número de Pessoas na Fila de Espera



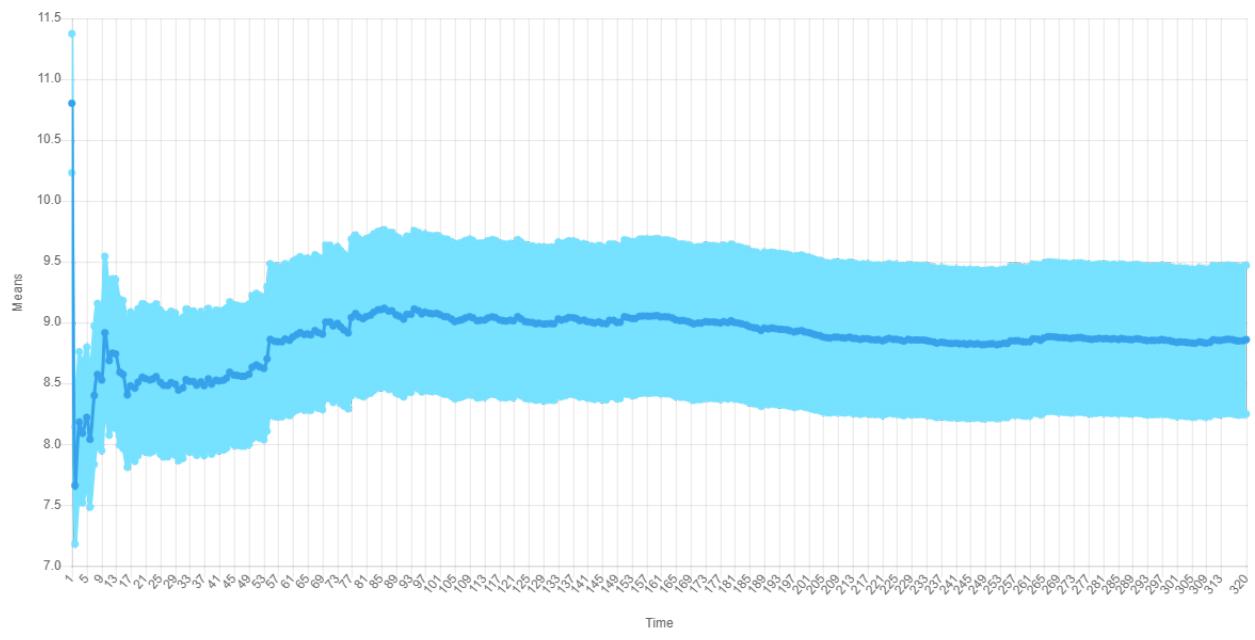
Variâncias do Número de Pessoas na Fila de Espera



Variância convergente: 18.8

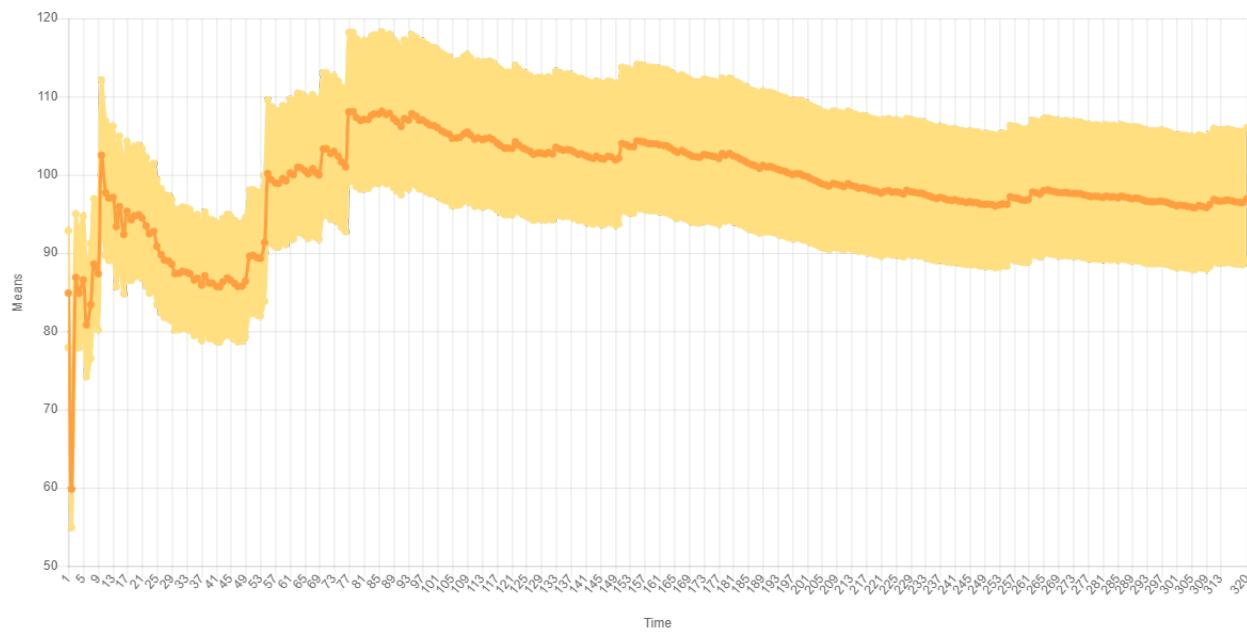
Utilização: 0.9

Médias dos Tempos de Espera em Fila



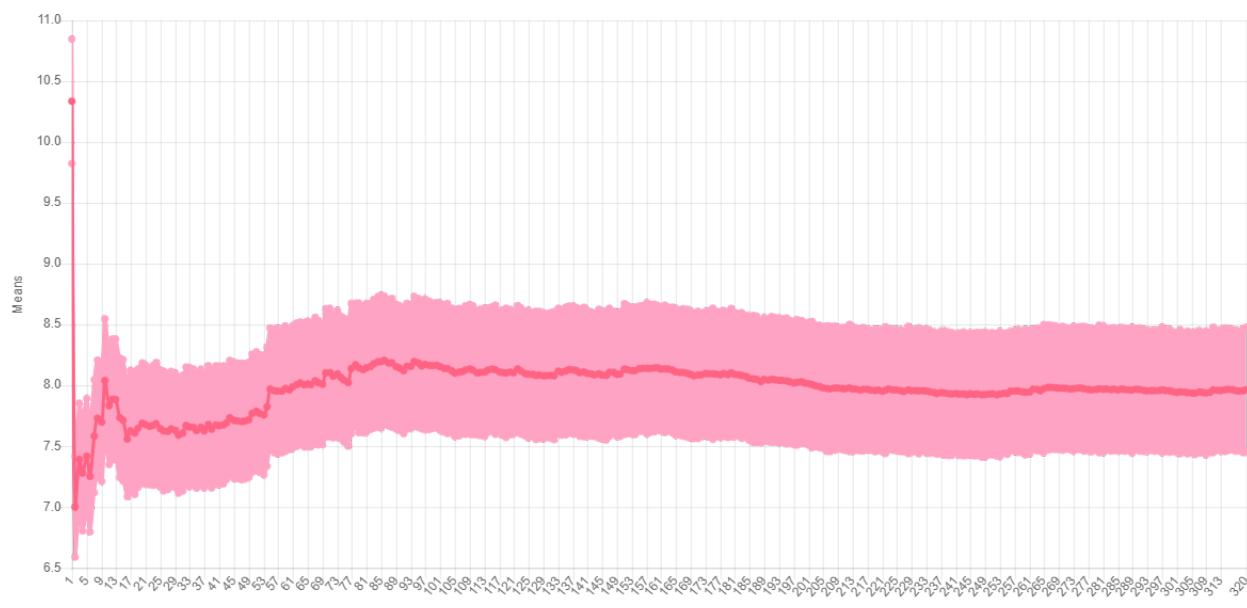
Média convergente: 9

Variâncias dos Tempos de Espera em Fila



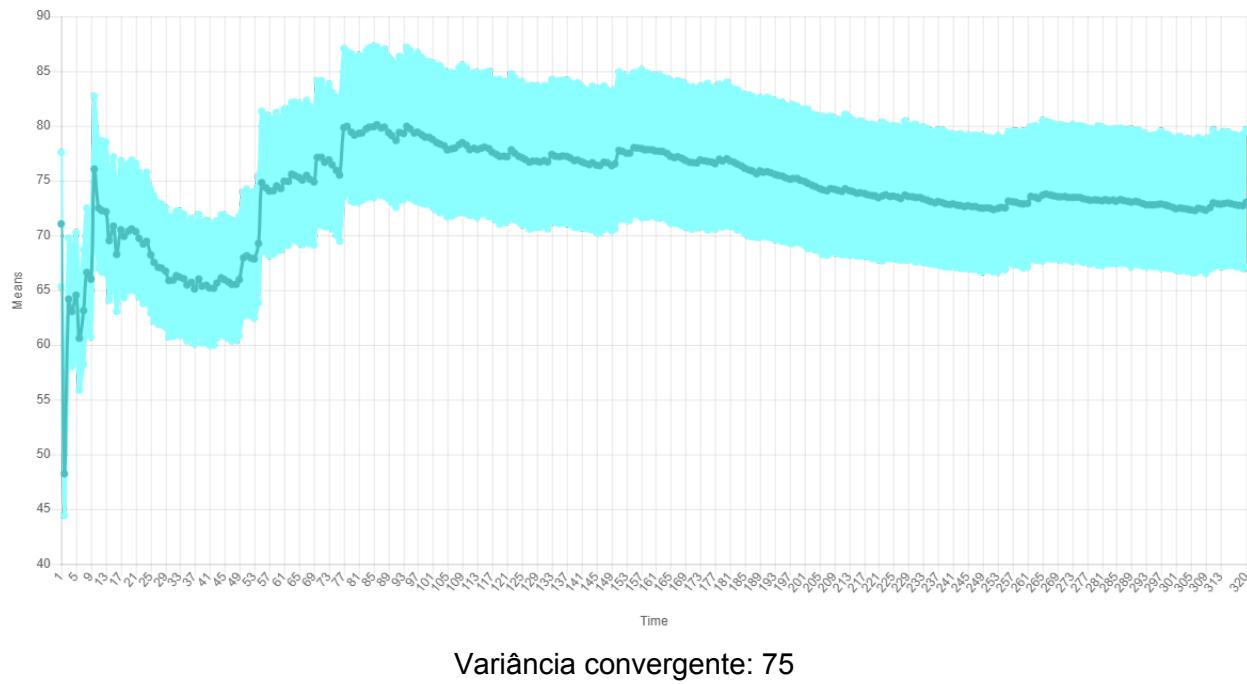
Variância convergente: 100

Médias do Número de Pessoas na Fila de Espera



Média convergente: 8

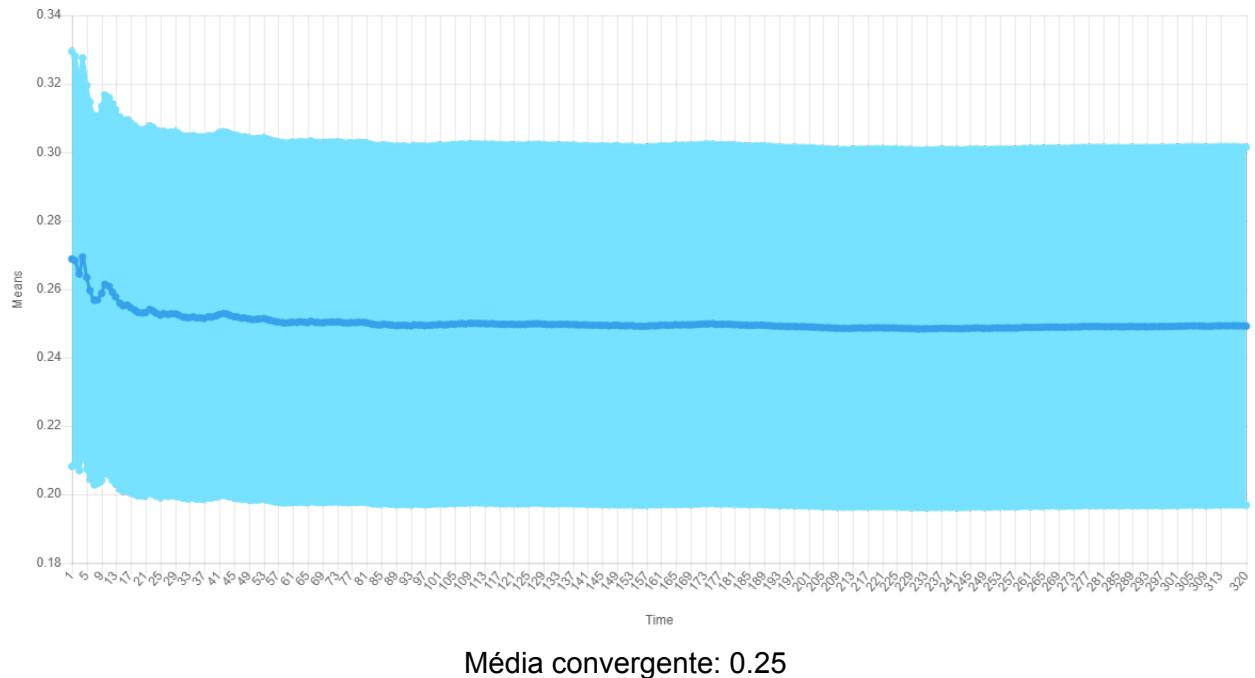
Variâncias do Número de Pessoas na Fila de Espera



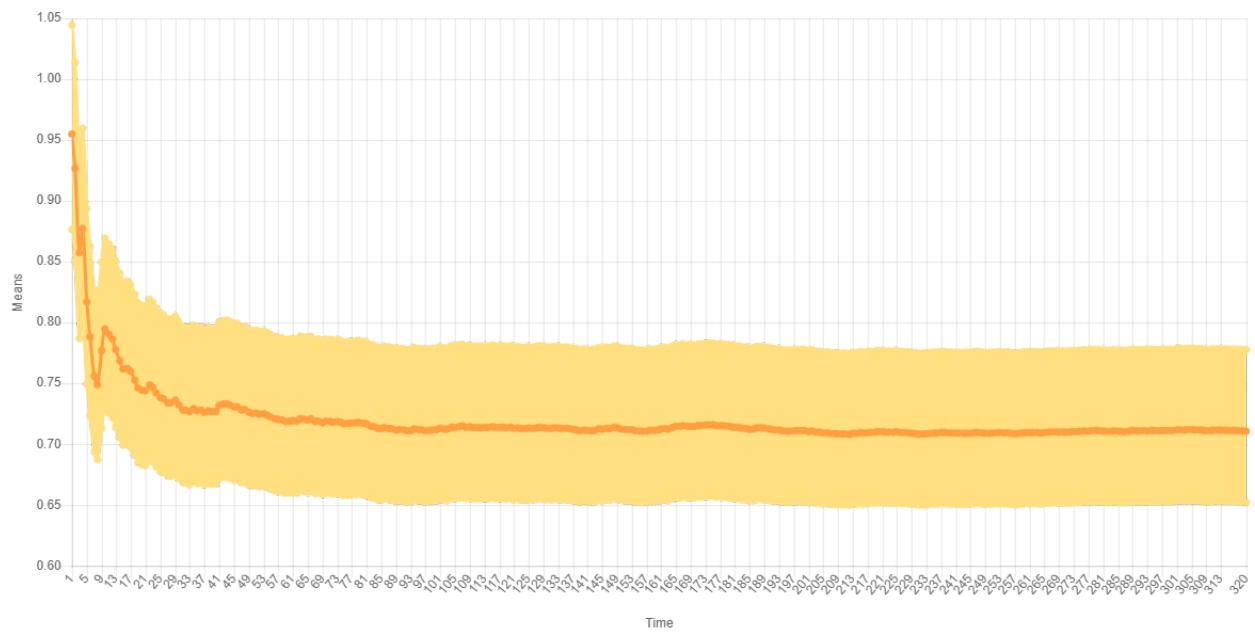
LCFS

Utilização: 0.2

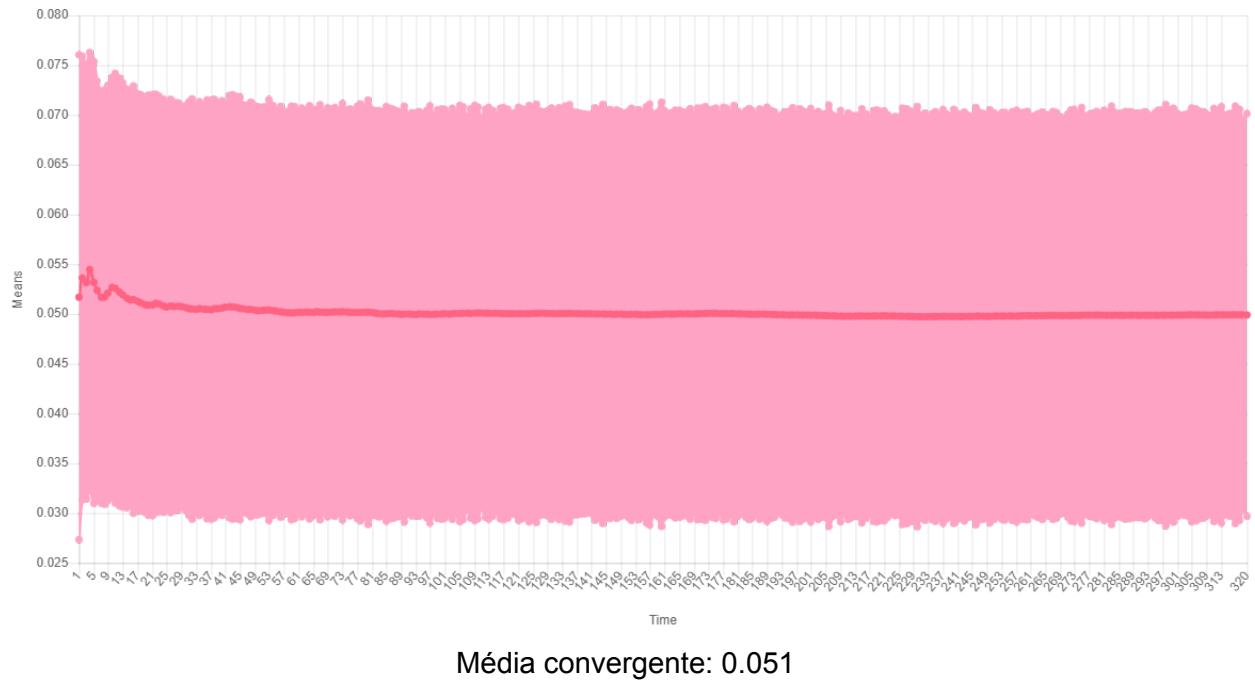
Médias dos Tempos de Espera em Fila



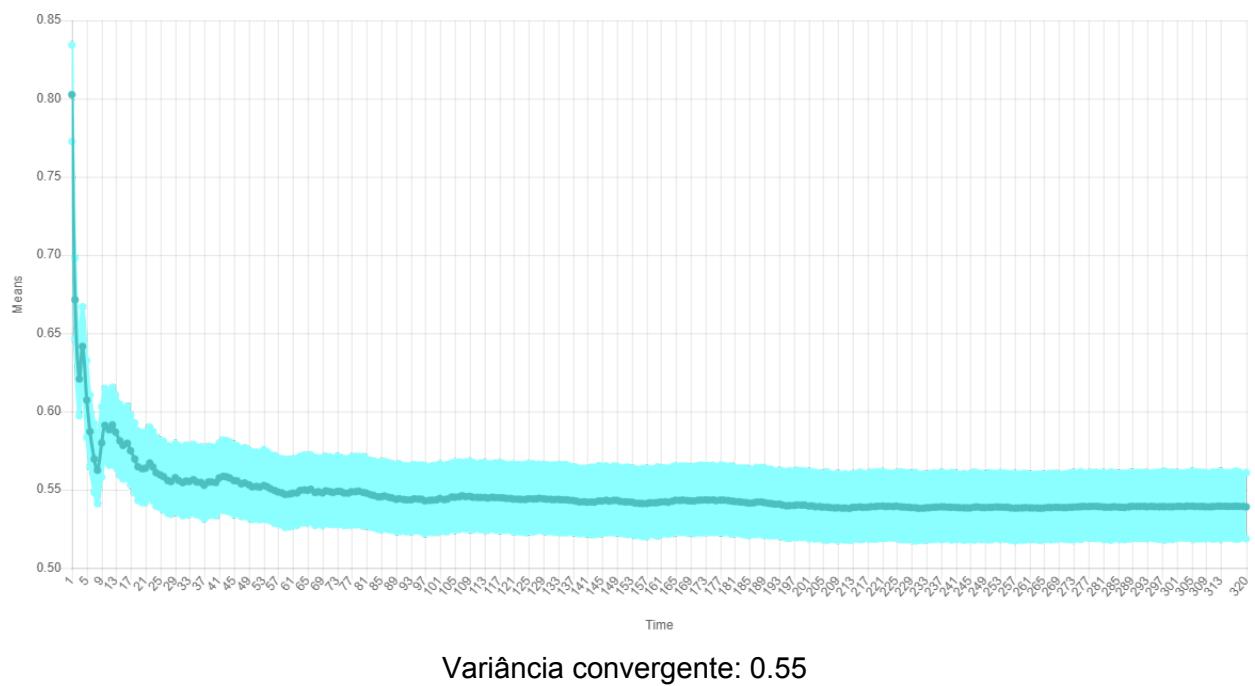
Variâncias dos Tempos de Espera em Fila



Médias do Número de Pessoas na Fila de Espera

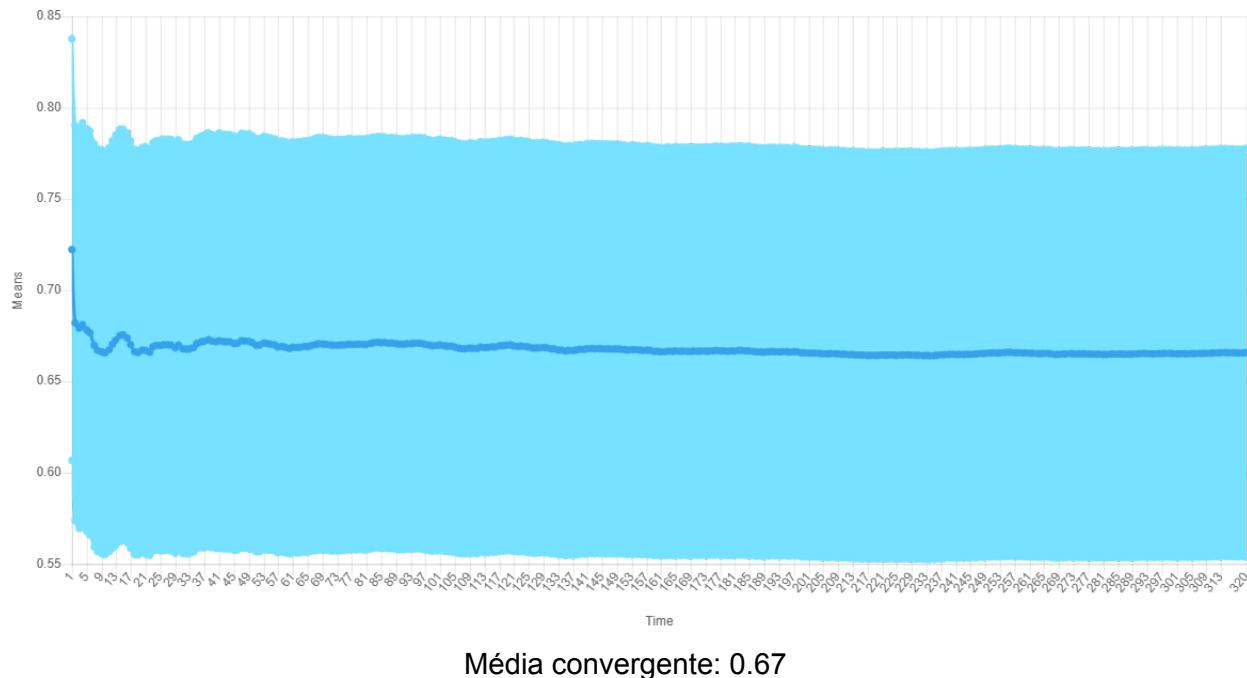


Variâncias do Número de Pessoas na Fila de Espera

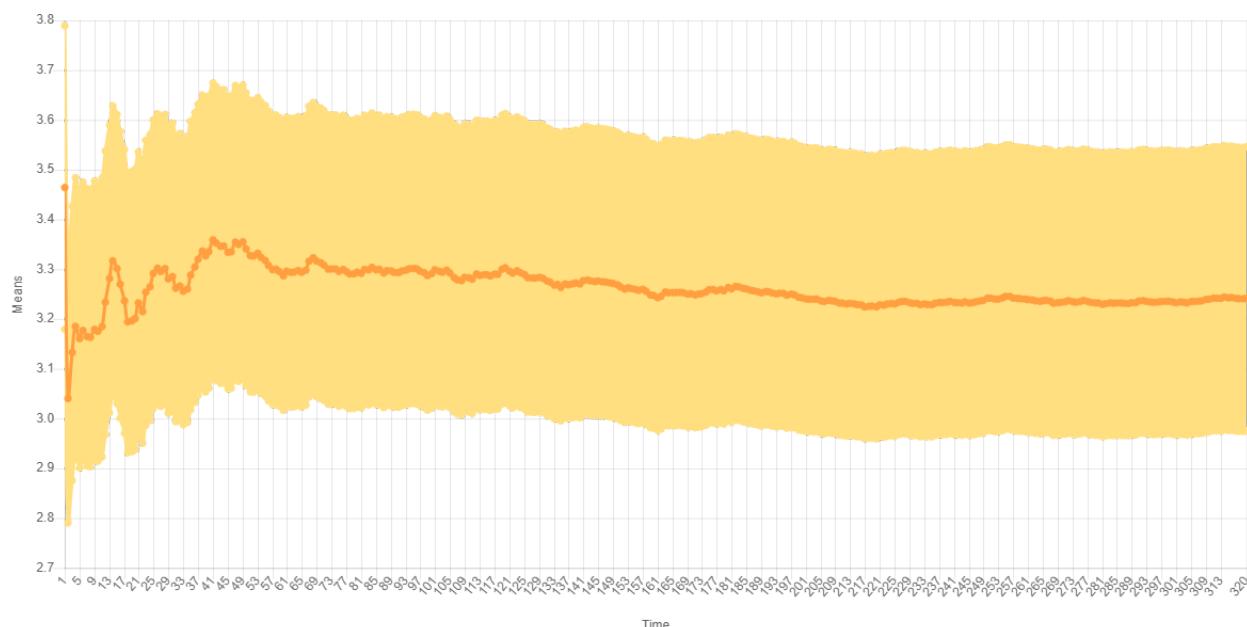


Utilização: 0.4

Médias dos Tempos de Espera em Fila

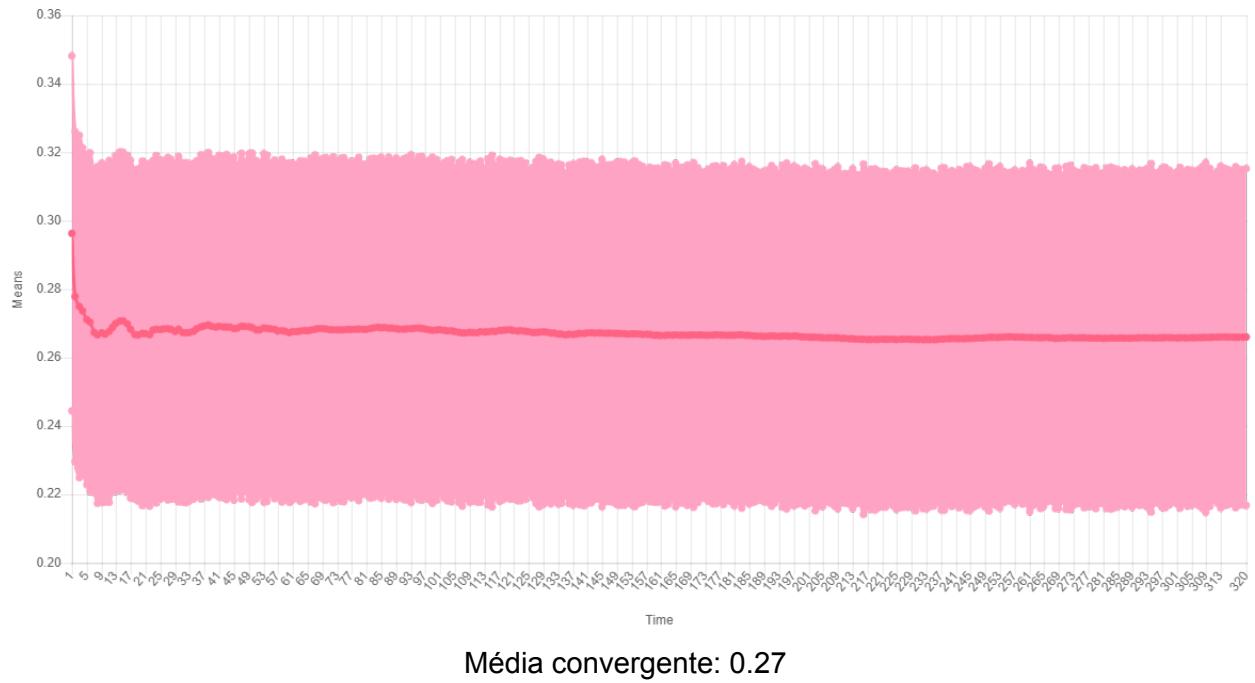


Variâncias dos Tempos de Espera em Fila

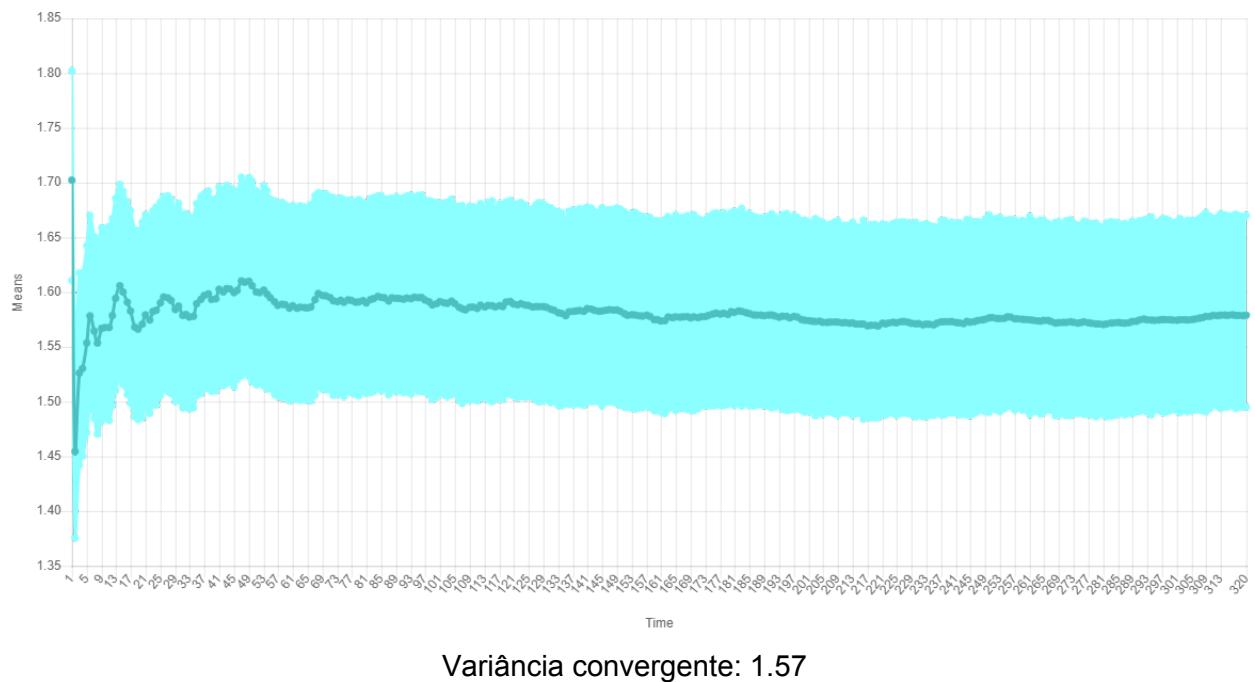


Variância convergente: 0.328

Médias do Número de Pessoas na Fila de Espera

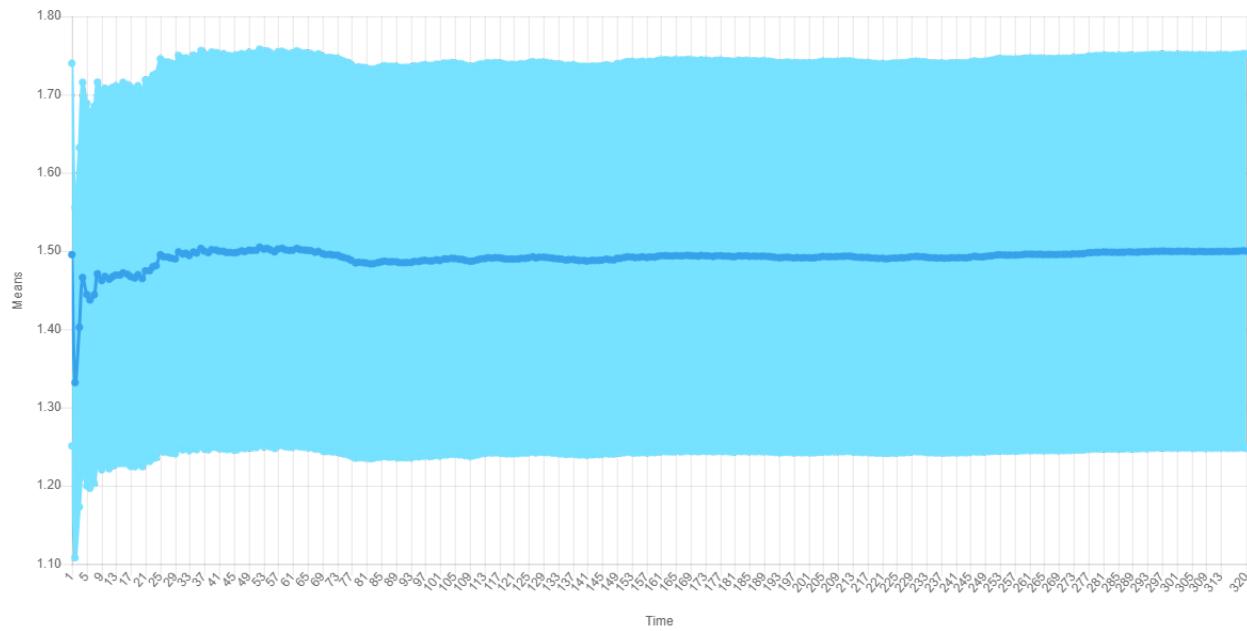


Variâncias do Número de Pessoas na Fila de Espera

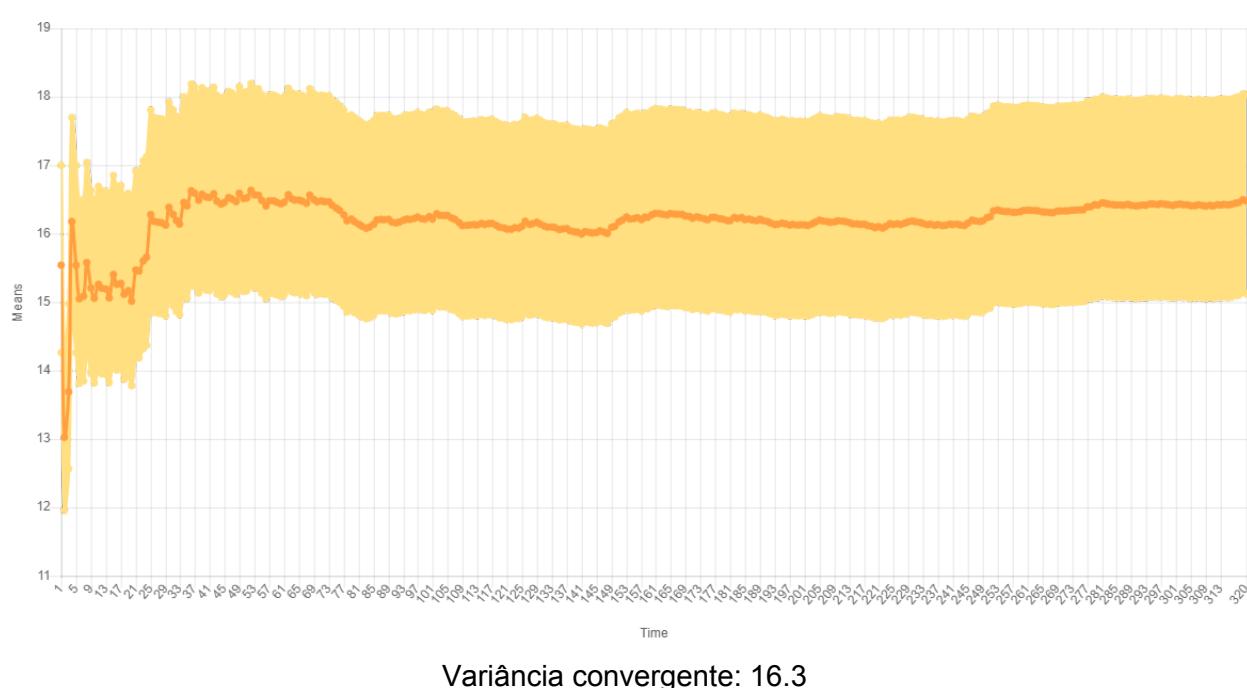


Utilização: 0.6

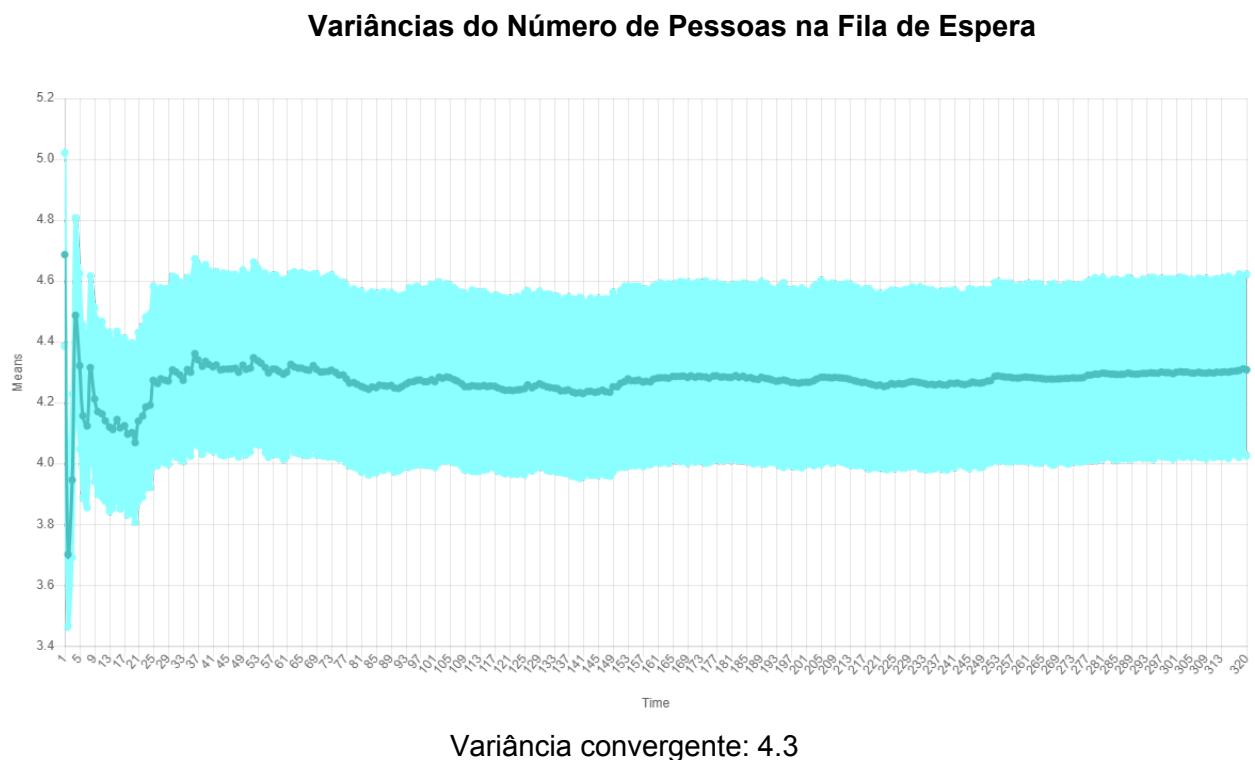
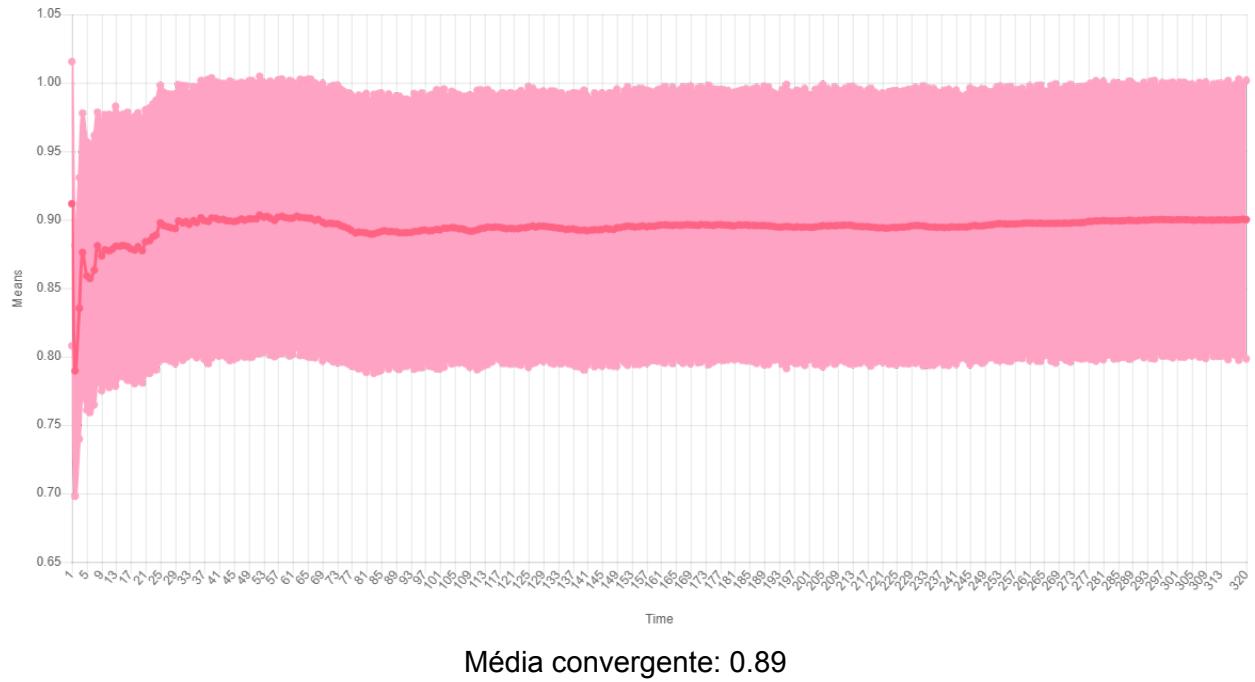
Médias dos Tempos de Espera em Fila



Variâncias dos Tempos de Espera em Fila

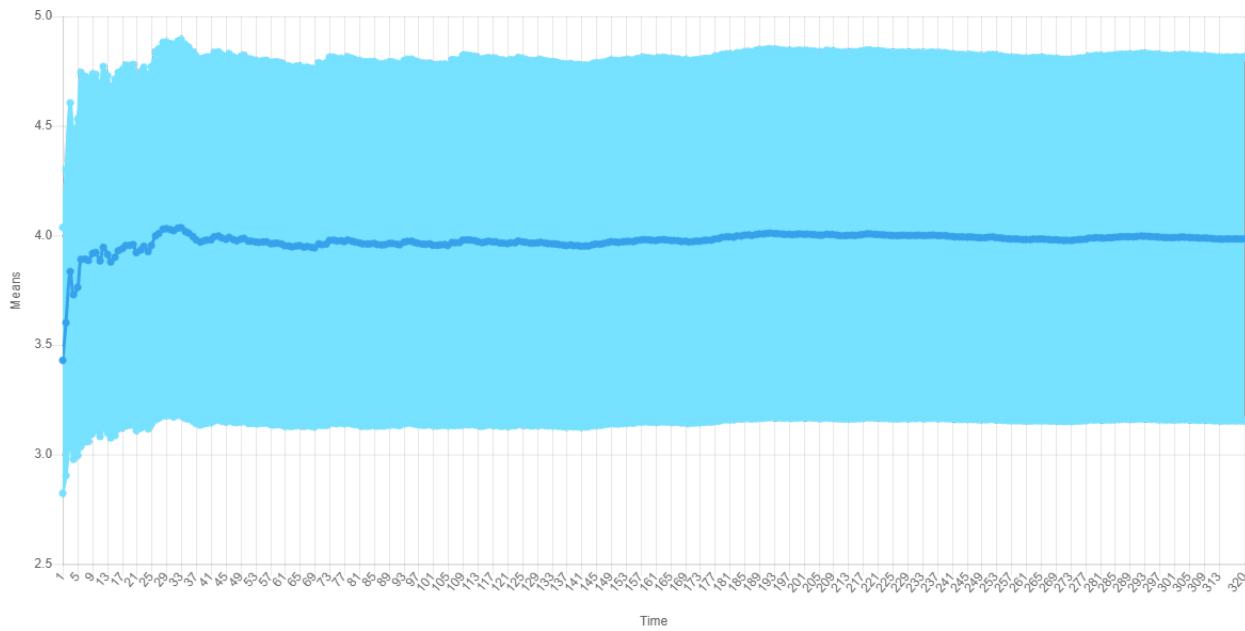


Médias do Número de Pessoas na Fila de Espera



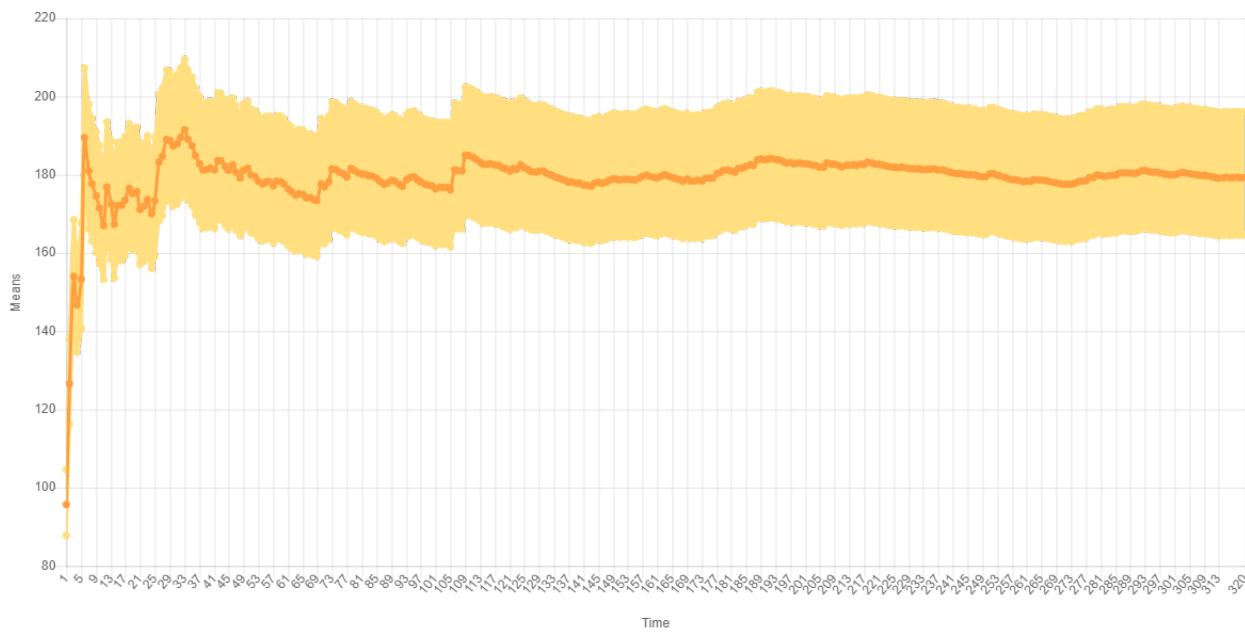
Utilização: 0.8

Médias dos Tempos de Espera em Fila



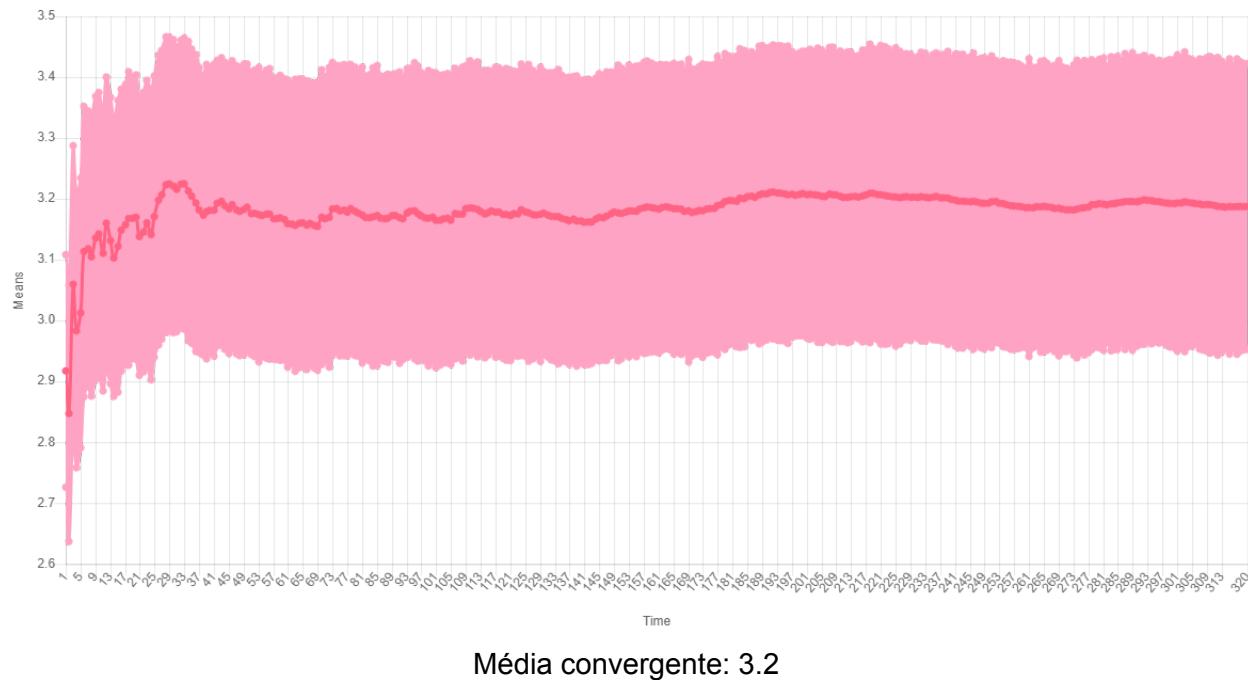
Média convergente: 4

Variâncias dos Tempos de Espera em Fila

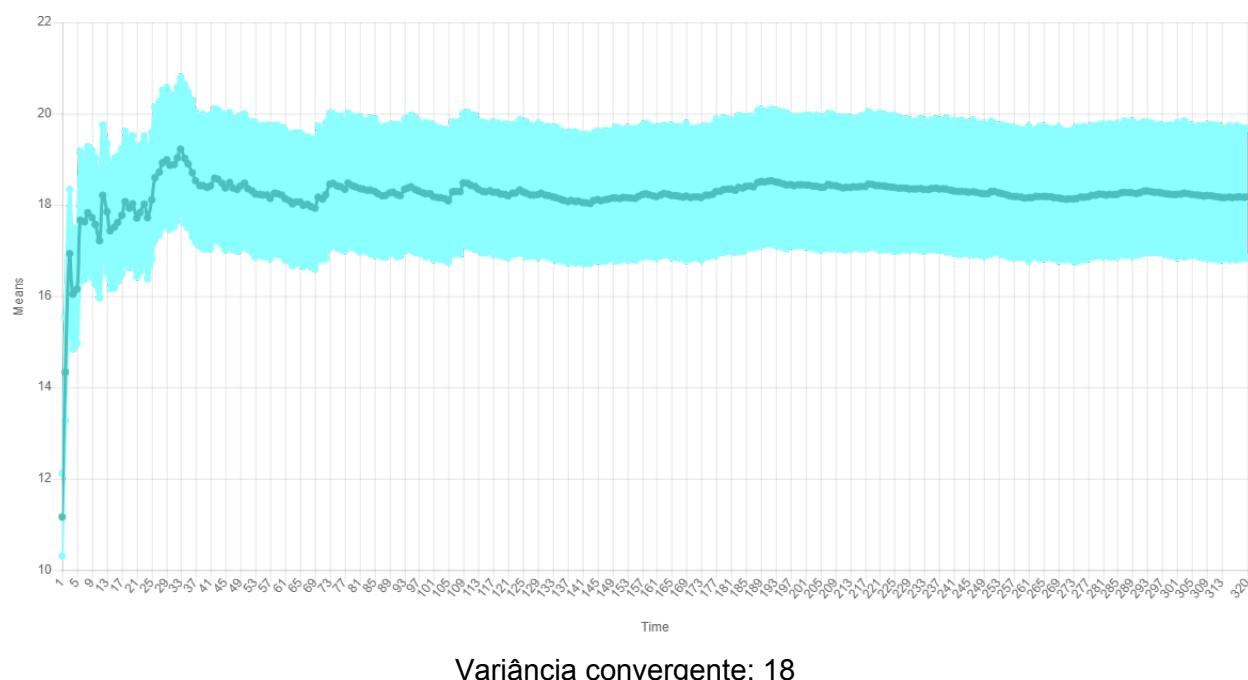


Variância convergente: 180

Médias do Número de Pessoas na Fila de Espera

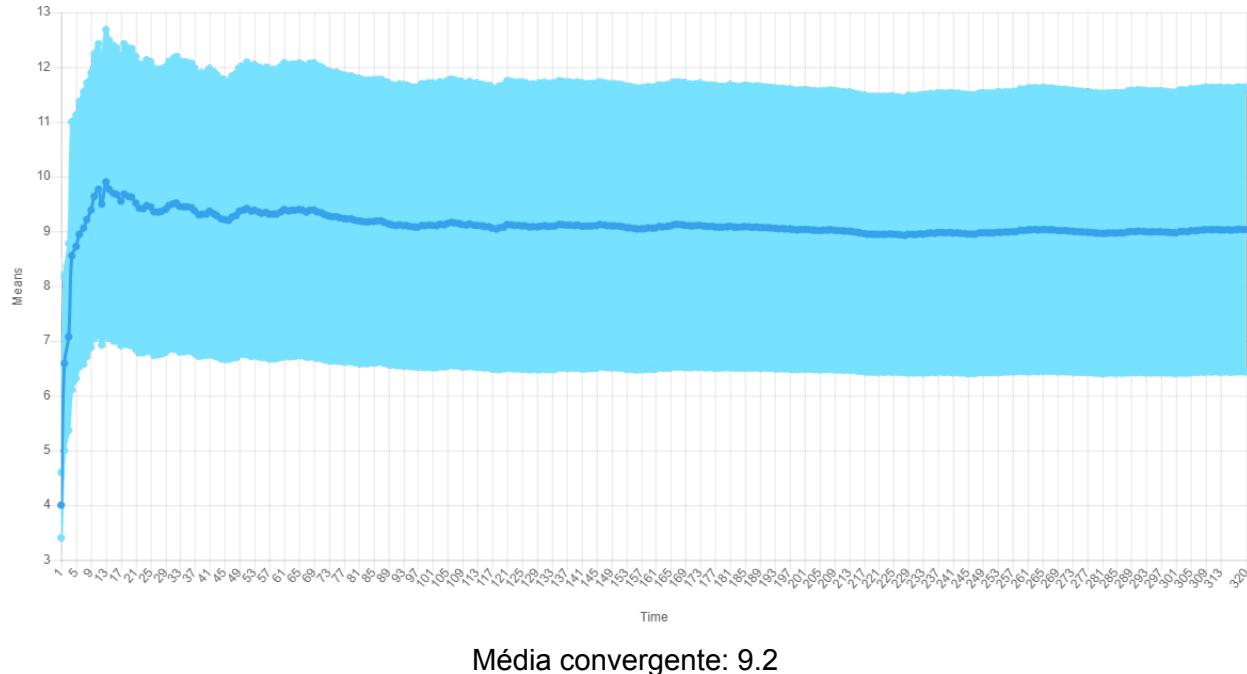


Variâncias do Número de Pessoas na Fila de Espera

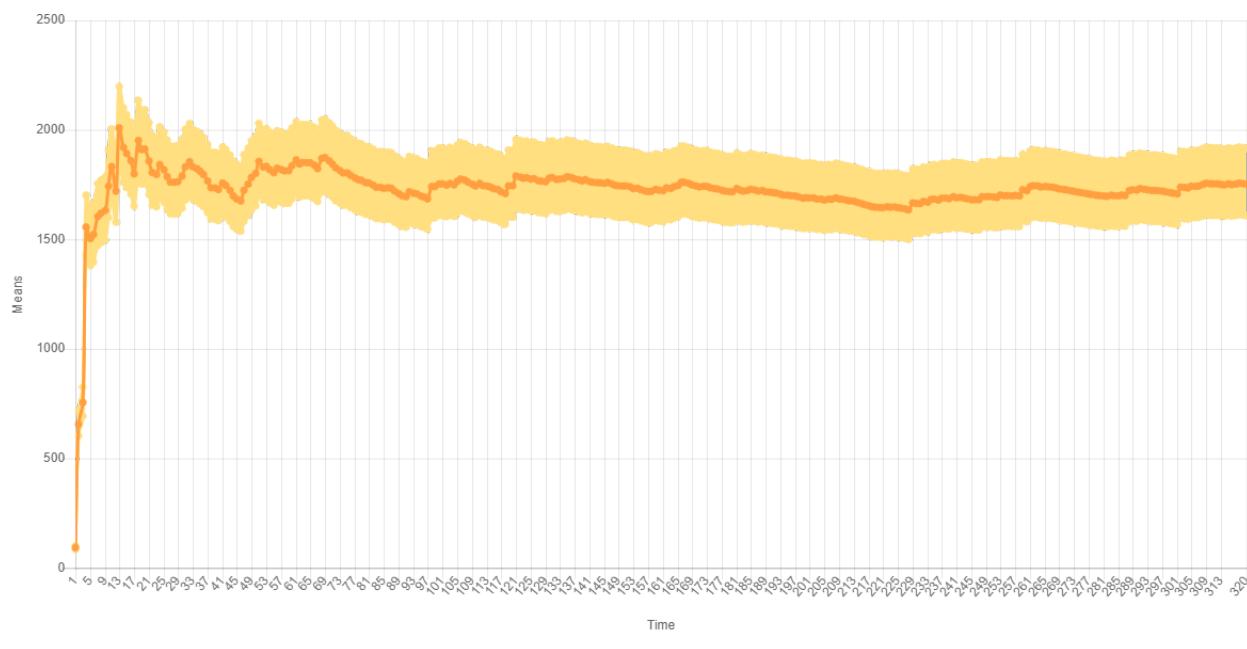


Utilização: 0.9

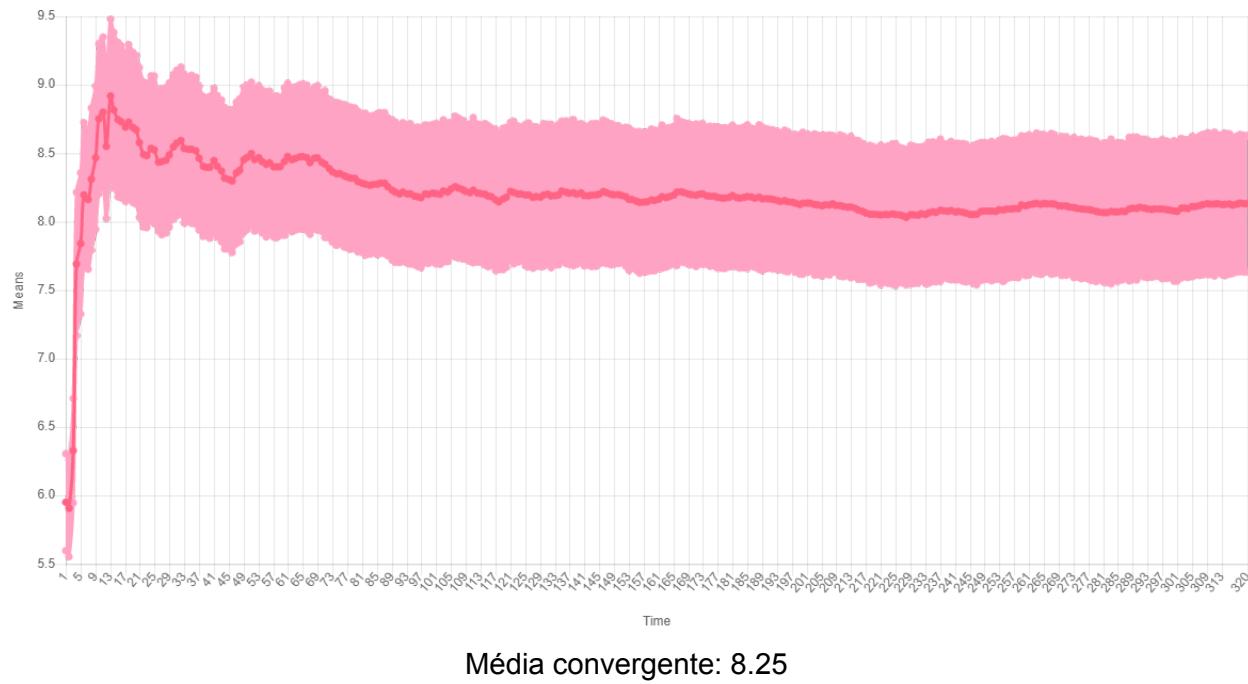
Médias dos Tempos de Espera em Fila



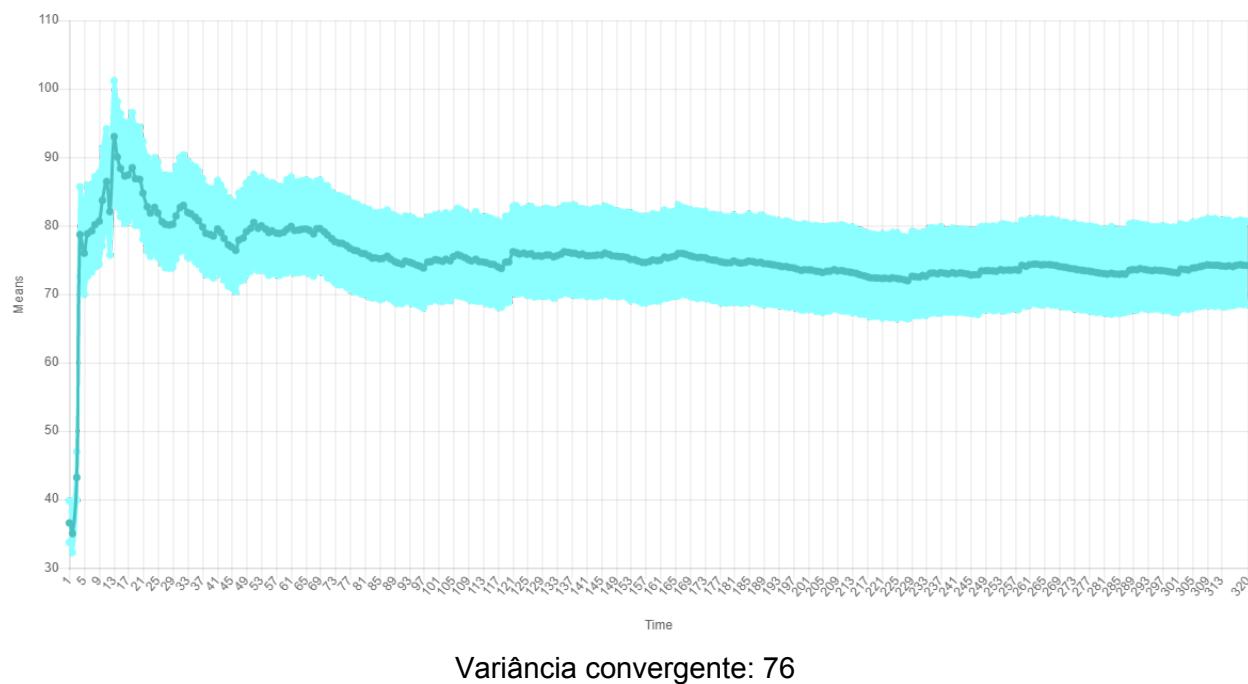
Variâncias dos Tempos de Espera em Fila



Médias do Número de Pessoas na Fila de Espera



Variâncias do Número de Pessoas na Fila de Espera



5) Conclusões

O trabalho, de um modo geral, foi interessante de fazer, por conta do escopo do problema. Em poucos momentos ao longo da graduação tivemos oportunidade de fazer um trabalho tão grande, em que várias classes precisam trabalhar em conjunto, assim como é no mercado de trabalho, então consideramos a realização deste trabalho importante para a nossa formação acadêmica e pro nosso preparo para o mercado.

Uma parte importante do desenvolvimento foi o fato de que utilizamos testes unitários para testar cada um dos métodos que foram programados. Estes testes estão disponíveis na pasta “tests”. Ao realizar os testes unitários, diversos erros na programação do código foram facilmente identificados e rapidamente corrigidos, o que evitou que esses erros se propagassem para outras classes. Isto diminuiu muito o esforço e o tempo gastos com *debug* do código, que se tornaria uma tarefa praticamente impossível se todos os erros se propagassem em conjunto.

Dentre as otimizações realizadas, nós procuramos manter poucas instâncias das classes ao longo da execução, de modo que a memória fosse ocupada apenas pelos objetos estritamente necessários. Esses objetos possuem suas referências apagadas tão logo não sejam mais necessários, e o coletor de lixo do Python realiza a liberação da área de memória que era utilizada por eles logo em seguida.

Em relação ao que pode ser melhorado, não conseguimos atingir o melhor *kmin* de coletas por rodada para a estimativa do intervalo de confiança da variância, porém estamos trabalhando para descobrir o problema e o corrigiremos o quanto antes possível.

6) Anexo - Documentação do Programa

O executável do código, junto com todo o código fonte, está disponível no GitHub, no link <https://github.com/brunobehnken/SimulatorMM1/>. O link para download direto da última versão disponível (referente a este relatório) é

https://github.com/brunobehnken/SimulatorMM1/releases/download/v0.2-beta/release_v0.2-beta.zip

Para rodar o código fonte, deve-se baixar o arquivo e descompactá-lo em um diretório. A seguir, é necessário abrir um terminal neste diretório e executar o arquivo *Simulator_v0.2-beta* utilizando o comando *./Simulator_v0.2-beta* ou o comando *bash Simulator_v0.2-beta*. O comando terá algumas linhas de texto de *output*. Uma delas informará o link *127.0.0.1:5000* (endereço de *loopback* na porta 5000). Esse endereço deve ser digitado na barra de endereços de qualquer navegador (enquanto isso o comando continuará rodando no terminal). Ao digitar o endereço e apertar *Enter*, o trabalho carregará na tela e poderá ser utilizado. É importante mencionar que o trabalho está disponível para execução apenas no sistema Linux.

A tela inicial do trabalho pode ser observada a seguir.



A seguir encontra-se o código fonte e a sua documentação.

```
class ExpGenerator:  
    """This class implements sample generators for the exponential distribution"""  
  
    def __init__(self, param_lambda=1, input_seed=None):  
        """Sets the lambda to be used by the generator  
        and the seed to be used by the RNG"""  
        self.__lambda = param_lambda  
        seed(input_seed)  
  
    def get_lambda(self):  
        """Returns the current value of lambda"""  
        return self.__lambda  
  
    def set_lambda(self, param_lambda):  
        """Set the value of lambda"""  
        self.__lambda = param_lambda  
  
    @staticmethod  
    def set_seed(input_seed):  
        """Set a new seed to be used by the RNG"""  
        seed(input_seed)  
  
    def get_exponential_time(self):  
        """Returns a sample time of the exponential  
        distribution using the current lambda"""  
        sample = random()  
        time = -1 * (1 / self.__lambda) * log(sample)  
        return time  
  
    def get_exponential_list(self, size):  
        """Returns a list of size 'size' of sample times of  
        the exponential distribution using the current lambda"""  
        time = []  
        for i in range(0, size):  
            sample = random()  
            res = -1 * (1 / self.__lambda) * log(sample)  
            time.append(res)  
        return time  
  
    @staticmethod  
    def get_exponential_time_lambda_1():  
        """Returns a sample time of the exponential  
        distribution using lambda = 1"""  
        sample = random()  
        time = -1 * log(sample)  
        return time
```

```
class Client:  
    """This class was made to represent a client that  
    arrives at the queue and is served by the server"""  
  
    def __init__(self, arrival_time, service_time):  
        """Sets the arrival time and the service time"""  
        self.__arrival_time = arrival_time  
        self.__service_time = service_time  
        self.__wait_time = None  
        self.__departure_time = None  
  
    def get_arrival_time(self):  
        """Returns the arrival time"""  
        return self.__arrival_time  
  
    def get_service_time(self):  
        """Returns the service time"""  
        return self.__service_time  
  
    def get_wait_time(self):  
        """Returns the wait time"""  
        return self.__wait_time  
  
    def set_wait_time(self, wait_time):  
        """Sets the wait time if possible"""  
        if self.__wait_time is None:  
            self.__wait_time = wait_time  
  
    def get_departure_time(self):  
        """Returns the departure time"""  
        return self.__departure_time  
  
    def set_departure_time(self, departure_time):  
        """Sets the departure time if possible"""  
        if self.__departure_time is None:  
            self.__departure_time = departure_time  
  
    def __str__(self):  
        return f"Arrival time: {self.__arrival_time}\n" +  
               f"Service time: {self.__service_time}\n" +  
               f"Wait time: {self.__wait_time}\n" +  
               f"Departure time: {self.__departure_time}\n"
```

```

# noinspection PyPep8Naming
class Master:

    @staticmethod
    def run_FCFS(rho, k):
        """Runs the simulation for FCFS discipline with given parameters,
        returning its statistics"""
        results_w = []
        results_nq = []
        stat_w = Statistics()
        stat_nq = Statistics()
        simulator = SimulatorFCFS(rho)
        simulator.transient_phase()
        for i in range(0, 3200):
            res = simulator.simulate_FCFS(k)
            mean_w = stat_w.calculate_incremental_mean(res[0])
            var_w = stat_w.calculate_incremental_variance(res[0])
            mean_nq = stat_nq.calculate_incremental_time_mean(res[1], res[2])
            var_nq = stat_nq.calculate_incremental_variance(res[1])
            center_ew, lower_ew, upper_ew, precision_ew = stat_w.confidence_interval_for_mean(mean_w, var_w,
                len(res[0]), 0.95)
            center_vw, lower_vw, upper_vw, precision_vw = stat_w.confidence_interval_for_variance(var_w, len(res[0]),
                0.95)
            center_enq, lower_enq, upper_enq, precision_enq = stat_nq.confidence_interval_for_mean(mean_nq, var_nq,
                res[2], 0.95)
            center_vnq, lower_vnq, upper_vnq, precision_vnq = stat_nq.confidence_interval_for_variance(var_nq,
                res[2], 0.95)
            results_w.append(((mean_w, center_ew, lower_ew, upper_ew, precision_ew),
                (var_w, center_vw, lower_vw, upper_vw, precision_vw)))
            results_nq.append(((mean_nq, center_enq, lower_enq, upper_enq, precision_enq),
                (var_nq, center_vnq, lower_vnq, upper_vnq, precision_vnq)))
        return results_w, results_nq

    @staticmethod
    def run_LCFS(rho, k):
        """Runs the simulation for LCFS discipline with given parameters,
        returning its statistics"""
        results_w = []
        results_nq = []
        stat_w = Statistics()
        stat_nq = Statistics()
        simulator = SimulatorLCFS(rho)
        simulator.transient_phase()
        for i in range(0, 3200):
            res = simulator.simulate_LCFS(k)
            mean_w = stat_w.calculate_incremental_mean(res[0])
            var_w = stat_w.calculate_incremental_variance(res[0])
            mean_nq = stat_nq.calculate_incremental_time_mean(res[1], res[2])
            var_nq = stat_nq.calculate_incremental_variance(res[1])
            center_ew, lower_ew, upper_ew, precision_ew = stat_w.confidence_interval_for_mean(mean_w, var_w,
                len(res[0]), 0.95)
            center_vw, lower_vw, upper_vw, precision_vw = stat_w.confidence_interval_for_variance(var_w, len(res[0]),
                0.95)
            center_enq, lower_enq, upper_enq, precision_enq = stat_nq.confidence_interval_for_mean(mean_nq, var_nq,
                res[2], 0.95)
            center_vnq, lower_vnq, upper_vnq, precision_vnq = stat_nq.confidence_interval_for_variance(var_nq,
                res[2], 0.95)
            results_w.append(((mean_w, center_ew, lower_ew, upper_ew, precision_ew),
                (var_w, center_vw, lower_vw, upper_vw, precision_vw)))
            results_nq.append(((mean_nq, center_enq, lower_enq, upper_enq, precision_enq),
                (var_nq, center_vnq, lower_vnq, upper_vnq, precision_vnq)))
        return results_w, results_nq

```

```

def webmain(self, discipline, rho):
    """Main function for running the simulator, to be called by Flask front-end.
    Returns the means and variances with all confidence intervals for both
    waiting times and number of client at the queue."""
    k = 1_000 # TODO this value is arbitrary for now but must be set later
    # k = 50 # TODO this value is arbitrary for now but must be set later
    discipline -= 1
    start_time = time()
    if discipline != 0 and discipline != 1:
        print("invalid input")
        return
    if discipline: # LCFS
        if rho == 0.2 or rho == 0.4 or rho == 0.6 or rho == 0.8 or rho == 0.9:
            print("Simulating...")
            results_w, results_nq = self.run_LCFS(rho, k)
        else:
            print("invalid input")
            return
    else: # FCFS
        if rho == 0.2 or rho == 0.4 or rho == 0.6 or rho == 0.8 or rho == 0.9:
            print("Simulating...")
            results_w, results_nq = self.run_FCFS(rho, k)
        else:
            print("invalid input")
            return
    end_time = time()
    print(f"Execution time: {end_time - start_time}")

    w_means, w_means_icl, w_means_icu, w_vars, w_vars_icl, w_vars_icu = [], [], [], [], [], []
    nq_means, nq_means_icl, nq_means_icu, nq_vars, nq_vars_icl, nq_vars_icu = [], [], [], [], [], []
    results_w = results_w[::10]
    results_nq = results_nq[::10]
    for i in range(len(results_w)):
        w_means.append(results_w[i][0][0])
        w_means_icl.append(results_w[i][0][2])
        w_means_icu.append(results_w[i][0][3])
        w_vars.append(results_w[i][1][0])
        w_vars_icl.append(results_w[i][1][2])
        w_vars_icu.append(results_w[i][1][3])
    for i in range(len(results_nq)):
        nq_means.append(results_nq[i][0][0])
        nq_means_icl.append(results_nq[i][0][2])
        nq_means_icu.append(results_nq[i][0][3])
        nq_vars.append(results_nq[i][1][0])
        nq_vars_icl.append(results_nq[i][1][2])
        nq_vars_icu.append(results_nq[i][1][3])

    return w_means, w_means_icl, w_means_icu, \
           w_vars, w_vars_icl, w_vars_icu, \
           nq_means, nq_means_icl, nq_means_icu, \
           nq_vars, nq_vars_icl, nq_vars_icu

```

```

class Scheduler:
    """This class implements a scheduler, which function is to manage
    time-ordered events that will take place in the simulator"""

    def __init__(self, param_lambda, seed=None):
        """Starts the scheduler and schedules the first arrival.
        'param_lambda' and 'seed' are used to generate exponential times"""

        self.__gen = ExpGenerator(param_lambda, seed)
        self.__last_arrival_time = 0

        # schedule is a time-ordered list of events that are represented as tuples.
        # Each tuple has 3 elements: the first is either 'a' (for an arrival event) or 'd' (for a departure event).
        # The second element is the time the event will take place (for simplicity of implementation only).
        # The third element is an instance of Client that represents
        # the client that will arrive to or depart from the system.
        self.__schedule = []

        # schedule first arrival
        client = Client(0, self.__gen.get_exponential_time_lambda_1())
        self.__update_schedule('a', client)

    def schedule_next_arrival(self):
        """Schedule the event of the next client arrival"""
        self.__last_arrival_time += self.__gen.get_exponential_time() # calculate next arrival time
        client = Client(self.__last_arrival_time, self.__gen.get_exponential_time_lambda_1())
        self.__update_schedule('a', client)

    def __update_schedule(self, event):
        """Updates the schedule with the event, preserving time-ordering"""
        i = 0
        size = len(self.__schedule)
        if event[0] == 'a':
            time = event[1].get_arrival_time()
            # seeks the right position to add the event to the schedule
            while i < size and self.__schedule[i][1] < time:
                i += 1
            self.__schedule.insert(i, ('a', time, event[1]))
        else:
            time = event[1].get_departure_time()
            # seeks the right position to add the event to the schedule
            while i < size and self.__schedule[i][1] < time:
                i += 1
            self.__schedule.insert(i, ('d', time, event[1]))

    def schedule_departure(self, client):
        """Schedule the event of the departure of the client currently being served.
        Assumes that wait_time and departure_time are already set by the server"""
        self.__update_schedule('d', client)

    def get_next_event(self):
        """Returns the next event in the schedule or None if the schedule is empty."""
        if len(self.__schedule) == 0:
            return None
        event = self.__schedule.pop(0)
        tup = (event[0], event[2])
        return tup

```

```

class SimulatorFCFS:
    """This class implements a FCFS queue simulator, which function is
    to gather statistics for this discipline of service"""

    def __init__(self, rho, seed=None):
        """Starts the simulator using 'rho' as its utilization.
        This parameter cannot be changed later"""
        self._rho = rho
        self._start_time = 0 # start_time of simulation rounds
        self._current_time = 0 # current time of the simulator (state variable kept over function calls)
        self._queue = [] # queue of the simulator (state variable kept over function calls)
        self._server_idle = True # state of the server of the simulator (state variable kept over function calls)
        self._scheduler = Scheduler(rho, seed) # Scheduler (state variable kept over function calls)
        self._waiting_times = [] # list of waiting times of the costumers, for statistics
        self._areas = [] # list of number of waiting costumers, for statistics

    # noinspection PyPep8Naming
    def simulate_FCFS(self, client_num):
        """Runs the simulation of a FCFS queue until 'client num' statistics are gathered,
        when the simulation is paused. The state of the simulation is kept to be used on future
        calls. Statistics are only relevant after transient phase has passed."""

        counter = 0 # Counter of wait_time statistics gathered
        self._start_time = self._current_time # start_time of this simulation round

        # Start Simulation
        next_event = self._scheduler.get_next_event() # next scheduled event
        while True: # keep simulating until break (when the necessary statistics are gathered)
            client = next_event[1] # client of the event being treated
            if next_event[0] == 'a': # if the event is an arrival
                delta_time = self._current_time # updates delta_time for statistics purposes
                self._current_time = client.get_arrival_time() # set current time to arrival time
                if self._server_idle: # if server is idle, client gets served immediately
                    client.set_wait_time(0) # no wait_time at all
                    self._areas.append(0) # updates 'waiting costumers' statistics with 0 waiting costumers
                    self._waiting_times.append(client.get_wait_time()) # updates wait time statistics
                    counter += 1 # Increments counter of wait_time statistics gathered
                    client.set_departure_time(client.get_arrival_time() + client.get_wait_time() +
                                              client.get_service_time()) # calculates departure time
                    self._scheduler.schedule_departure(client) # schedule client departure event
                    self._server_idle = False # server is now busy serving the client
                else: # if server is busy, client goes to the queue
                    if self._queue: # if queue is not empty, updates 'waiting costumers' statistics
                        delta_time -= self._current_time # calculates the amount of time
                        delta_time *= -1 # fix delta_time sign
                        area = delta_time * len(self._queue) # calculates the area under the graphic
                        self._areas.append(area) # updates 'waiting costumers' statistics
                    self._queue.append(client) # client goes to the queue
            self._scheduler.schedule_next_arrival() # schedule next arrival
            if counter == client_num: # if the requested statistics were gathered, pause simulation
                break # ATTENTION: simulation should NOT break before schedule_next_arrival()

```

```

        else: # if the event is a departure
            delta_time = self._current_time # updates delta_time for statistics purposes
            self._current_time = client.get_departure_time() # set current time to departure time
            del client # client has departed :
            if self._queue: # if queue is not empty, serve next client
                delta_time -= self._current_time # calculates the amount of time
                delta_time *= -1 # fix delta_time sign
                area = delta_time * len(self._queue) # calculates the area under the graphic
                self._areas.append(area) # updates waiting costumers' statistics
                client = self._queue.pop(0) # call the next client using FCFS discipline
                client.set_wait_time(self._current_time - client.get_arrival_time()) # calculates the wait_time
                self._waiting_times.append(client.get_wait_time()) # updates wait_time statistics
                counter += 1 # Increments counter of wait_time statistics gathered
                client.set_departure_time(client.get_arrival_time() + client.get_wait_time() +
                                         client.get_service_time()) # calculates departure time
                self._scheduler.schedule_departure(client) # schedule client departure event
                if counter == client_num: # if the requested statistics were gathered, pause simulation
                    break
            else: # if there is no next client, idle until next arrival
                self._server_idle = True
        next_event = self._scheduler.get_next_event() # next scheduled event

    final_waiting_times = self._waiting_times[:] # creates copy of waiting_times list to return
    self._waiting_times.clear() # clears the waiting_times list for next simulation round
    final_areas = self._areas[:] # creates copy of areas list to return
    self._areas.clear() # clears the areas list for next simulation round
    round_runtime = self._current_time - self._start_time # calculates amount of time of this simulation round
    res = (final_waiting_times, final_areas, round_runtime) # builds the return tuple
    return res

def transient_phase(self):
    """Runs the simulator until the transient phase is finished, which occurs
    when diff_limit variance values under threshold precision are found"""
    stats = Statistics()
    means_w = []
    old_var = 0
    diff_counter = 0

    if self._rho == 0.8 or self._rho == 0.9:
        threshold = 1.0e-01
        diff_limit = 150
    else:
        threshold = 1.0e-04
        diff_limit = 30
    while True:
        for i in range(0, 10):
            res = self.simulate_FCFS(100)
            means_w.append(stats.calculate_mean(res[0]))
            var_w = stats.calculate_incremental_variance(means_w)
            means_w.clear()
            diff = abs(var_w - old_var)
            old_var = var_w
            if diff < threshold:
                diff_counter += 1
                if diff_counter == diff_limit:
                    break

```

```

class SimulatorLCFS:
    """This class implements a LCFS queue simulator, which function is
    to gather statistics for this discipline of service"""

    def __init__(self, rho, seed=None):
        """Starts the simulator using 'rho' as its utilization.
        This parameter cannot be changed later"""
        self._rho = rho
        self._start_time = 0 # start_time of simulation rounds
        self._current_time = 0 # current time of the simulator (state variable kept over function calls)
        self._queue = [] # queue of the simulator (state variable kept over function calls)
        self._server_idle = True # state of the server of the simulator (state variable kept over function calls)
        self._scheduler = Scheduler(rho, seed) # Scheduler (state variable kept over function calls)
        self._waiting_times = [] # list of waiting times of the costumers, for statistics
        self._areas = [] # list of number of waiting costumers, for statistics

    # noinspection PyPep8Naming
    def simulate_LCFS(self, client_num):
        """Runs the simulation of a LCFS queue until 'client_num' statistics are gathered,
        when the simulation is paused. The state of the simulation is kept to be used on future
        calls. Statistics are only relevant after transient phase has passed."""

        counter = 0 # Counter of wait_time statistics gathered
        self._start_time = self._current_time # start_time of this simulation round

        # Start Simulation
        next_event = self._scheduler.get_next_event() # next scheduled event
        while True: # keep simulating until break (when the necessary statistics are gathered)
            client = next_event[1] # client of the event being treated
            if next_event[0] == 'a': # if the event is an arrival
                delta_time = self._current_time # updates delta_time for statistics purposes
                self._current_time = client.get_arrival_time() # set current time to arrival time
                if self._server_idle: # if server is idle, client gets served immediately
                    client.set_wait_time(0) # no wait time at all
                    self._areas.append(0) # updates 'waiting costumers' statistics with 0 waiting costumers
                    self._waiting_times.append(client.get_wait_time()) # updates wait_time statistics
                    counter += 1 # Increments counter of wait_time statistics gathered
                    client.set_departure_time(client.get_arrival_time() + client.get_wait_time() +
                                              client.get_service_time()) # calculates departure time
                    self._scheduler.schedule_departure(client) # schedule client departure event
                    self._server_idle = False # server is now busy serving the client
                else: # if server is busy, client goes to the queue
                    if self._queue: # if queue is not empty, updates 'waiting costumers' statistics
                        delta_time -= self._current_time # calculates the amount of time
                        delta_time *= -1 # fix delta_time sign
                        area = delta_time * len(self._queue) # calculates the area under the graphic
                        self._areas.append(area) # updates 'waiting costumers' statistics
                    self._queue.append(client) # client goes to the queue
            self._scheduler.schedule_next_arrival() # schedule next arrival
            if counter == client_num: # if the requested statistics were gathered, pause simulation
                break # ATTENTION: simulation should NOT break before schedule_next_arrival()

```

```

    else: # if the event is a departure
        delta_time = self._current_time # updates delta_time for statistics purposes
        self._current_time = client.get_departure_time() # set current time to departure time
        del client # client has departed :
        if self._queue: # if queue is not empty, serve next client
            delta_time -= self._current_time # calculates the amount of time
            delta_time *= -1 # fix delta_time sign
            area = delta_time * len(self._queue) # calculates the area under the graphic
            self._areas.append(area) # updates 'waiting costumers' statistics
            client = self._queue.pop() # call the next client using LCFS discipline
            client.set_wait_time(self._current_time - client.get_arrival_time()) # calculates the wait_time
            self._waiting_times.append(client.get_wait_time()) # updates wait_time statistics
            counter += 1 # Increments counter of wait_time statistics gathered
            client.set_departure_time(client.get_arrival_time() + client.get_wait_time() +
                                      client.get_service_time()) # calculates departure time
            self._scheduler.schedule_departure(client) # schedule client departure event
            if counter == client_num: # if the requested statistics were gathered, pause simulation
                break
        else: # if there is no next client, idle until next arrival
            self._server_idle = True
    next_event = self._scheduler.get_next_event() # next scheduled event

    final_waiting_times = self._waiting_times[:] # creates copy of waiting times list to return
    self._waiting_times.clear() # clears the waiting_times list for next simulation round
    final_areas = self._areas[:] # creates copy of areas list to return
    self._areas.clear() # clears the areas list for next simulation round
    round_runtime = self._current_time - self._start_time # calculates amount of time of this simulation round
    res = (final_waiting_times, final_areas, round_runtime) # builds the return tuple
    return res

def transient_phase(self):
    """Runs the simulator until the transient phase is finished, which occurs
    when 'diff_limit' delta variance values under 'threshold' precision are found"""
    stats = Statistics()
    means_w = []
    old_var = 0
    diff_counter = 0

    if self._rho == 0.8 or self._rho == 0.9:
        threshold = 1.0e-01
        diff_limit = 150
    else:
        threshold = 1.0e-04
        diff_limit = 30
    while True:
        for i in range(0, 10):
            res = self.simulate_LCFS(100)
            means_w.append(stats.calculate_mean(res[0]))
        var_w = stats.calculate_incremental_variance(means_w)
        means_w.clear()
        diff = abs(var_w - old_var)
        old_var = var_w
        if diff < threshold:
            diff_counter += 1
            if diff_counter == diff_limit:
                break

```

```
class Statistics:

    def __init__(self):
        self.__mean_sample_sum = 0
        self.__mean_num_sample = 0
        self.__var_sample_sum = 0
        self.__var_sample_sum_of_squares = 0
        self.__var_num_sample = 0
        self.__time_mean_sample_sum = 0
        self.__time_mean_num_sample = 0

    @staticmethod
    def calculate_mean(sample):
        """Returns a float with the mean of a sample"""
        return sum(sample) / len(sample)

    def calculate_incremental_mean(self, sample):
        """Returns a float with the mean of all the samples given until now.
        Values can be reset using the reset method."""
        self.__mean_sample_sum += sum(sample)
        self.__mean_num_sample += len(sample)
        return self.__mean_sample_sum / self.__mean_num_sample

    def calculate_incremental_time_mean(self, sample, length):
        """Returns a float with the mean of all the samples given until now
        using the length as denominator.
        Values can be reset using the reset method."""
        self.__time_mean_sample_sum += sum(sample)
        self.__time_mean_num_sample += length
        return self.__time_mean_sample_sum / self.__time_mean_num_sample

    @staticmethod
    def calculate_variance(sample, mean):
        """Returns a float with the variance of a sample"""
        variance_sum = 0
        for _sample in sample:
            variance_sum += (_sample - mean) ** 2
        return variance_sum / (len(sample) - 1)

    def calculate_incremental_variance(self, sample):
        """Returns a float with the variance of all the samples given until now.
        Values can be reset using the reset method."""
        self.__var_sample_sum += sum(sample)
        self.__var_num_sample += len(sample)
        for i in range(0, len(sample)):
            self.__var_sample_sum_of_squares += sample[i] ** 2
        return (self.__var_sample_sum_of_squares / (self.__var_num_sample - 1))\
            - (self.__var_sample_sum ** 2 / (self.__var_num_sample * (self.__var_num_sample - 1)))
```

```
@staticmethod
def confidence_interval_for_variance(variance, num_rounds, confidence_interval):
    """Calculates interval of confidence for the variance by the Chi-square distribution
    Returns the center of the interval, the upper and lower bounds and its precision"""
    alpha_lower = 1 - confidence_interval
    alpha_lower /= 2
    alpha_upper = 1 - alpha_lower
    df = num_rounds - 1
    chi2_lower = chi2.isf(alpha_lower, df)
    chi2_upper = chi2.isf(alpha_upper, df)
    lower = df * variance / chi2_lower
    upper = df * variance / chi2_upper
    precision = (chi2_lower - chi2_upper) / (chi2_lower + chi2_upper)
    center = (upper + lower) / 2
    return center, lower, upper, precision

def reset(self):
    """This method resets the metrics for incremental calculus of mean and variance"""
    self._mean_sample_sum = 0
    self._mean_num_sample = 0
    self._var_sample_sum = 0
    self._var_sample_sum_of_squares = 0
    self._var_num_sample = 0
    self._time_mean_sample_sum = 0
    self._time_mean_num_sample = 0
```

```

app = Flask(__name__)

@app.route("/", methods=['GET', 'POST'])
def home():
    discipline = None
    rho = None
    if request.method == 'POST':
        if request.form['discipline'] == 'fcfs':
            discipline = 1
        elif request.form['discipline'] == 'lcfs':
            discipline = 2
        if request.form['util'] == 'util2':
            rho = 0.2
        elif request.form['util'] == 'util4':
            rho = 0.4
        elif request.form['util'] == 'util6':
            rho = 0.6
        elif request.form['util'] == 'util8':
            rho = 0.8
        elif request.form['util'] == 'util9':
            rho = 0.9
    return redirect(url_for('simul', discipline=discipline, rho=rho))

    elif request.method == 'GET':
        return render_template("home.html")

@app.route("/simul/disc<int:discipline>util<float:rho>")
def simul(discipline, rho):
    master = Master()
    results_w, results_w_icl, results_w_icu, \
        results_w_vars, results_w_vars_icl, results_w_vars_icu, \
        results_nq, results_nq_icl, results_nq_icu, \
        results_nq_vars, results_nq_vars_icl, results_nq_vars_icu = master.webmain(discipline, rho)
    discipline = "FCFS" if discipline == 1 else "LCFS"
    return render_template("simulation.html", discipline=discipline, rho=rho,
        results_w=json.dumps(results_w),
        results_w_icl=json.dumps(results_w_icl),
        results_w_icu=json.dumps(results_w_icu),
        results_w_vars=json.dumps(results_w_vars),
        results_w_vars_icl=json.dumps(results_w_vars_icl),
        results_w_vars_icu=json.dumps(results_w_vars_icu),
        results_nq=json.dumps(results_nq),
        results_nq_icl=json.dumps(results_nq_icl),
        results_nq_icu=json.dumps(results_nq_icu),
        results_nq_vars=json.dumps(results_nq_vars),
        results_nq_vars_icl=json.dumps(results_nq_vars_icl),
        results_nq_vars_icu=json.dumps(results_nq_vars_icu))

if __name__ == "__main__":
    app.run(debug=True)

```