Avaliação Comparativa de Desempenho

Equipe 6:
Allan Soares Vasconcelos
Kennedy Edmilson Cunha Melo
Leilany Alves Aragao Ulisses
Rafael dos Reis de Labio

Centro de Informática UFPE
Universidade Federal de Pernambuco

Passo 1

Objetivo

Passo 1: Objetivo

Comparar o desempenho (tempo de execução) das duas implementações da aplicação cliente-servidor, que simula um sistema de mensagens baseados em filas, um feito com RabbitMQ e outro com nosso próprio algoritmo gerenciador de filas (Simplified).

A métrica-chave será o tempo médio decorrido entre a publicação de uma mensagem pelo produtor e seu recebimento pelo consumidor (latência).

Passo 2

Serviços do Sistema

Passo 2: Serviços do Sistema

- Para a implementação simplificada:
  - Envio de mensagens pelos produtores para o servidor de fila
  - Armazenamento das mensagens na fila FIFO em memória
  - Disponibilização das mensagens em ordem FIFO para os consumidores
  - Manipulação de solicitações PUSH (inserção na fila)
  - Manipulação de solicitações PULL (consumo da fila)
  - Gerenciamento de conexões TCP concorrentes

consumer.go 2, M

~/progConcorrente/atividade4-distribuidos/simplified/consumer.go (preview)

```go
}

func main() {
    addr := flag.String("addr", "localhost:9000", "queue server <host:port>")
    n := flag.Int("n", 1000, "number of messages to expect before exiting")
    flag.Parse()

    conn, err := net.Dial("tcp", *addr)
    if err != nil {
        log.Fatalf("dial: %v", err)
    }
    defer conn.Close()
    reader := bufio.NewReader(conn)      You, 8 minutes ago • Uncommitted change
    writer := bufio.NewWriter(conn)

    var latencies []time.Duration
    for len(latencies) < *n {
        // request next message
        fmt.Fprintf(writer, "PULL\n")
        writer.Flush()

        line, err := reader.ReadString('\n')
        if err != nil {
            log.Fatal(err)
        }
        line = strings.TrimSpace(line)

        switch {
        case strings.HasPrefix(line, "MSG "):
            tsStr := strings.TrimPrefix(line, "MSG ")
            sent, _ := strconv.ParseInt(tsStr, 10, 64)
            lat := time.Now().UnixNano() - sent
            latencies = append(latencies, time.Duration(lat))
        case line == "EMPTY":
            time.Sleep(100 * time.Microsecond) // brief back-off
        default:
            log.Printf("unexpected: %q", line)
        }
    }

    // simple stats
    var sum time.Duration
    for _, l := range latencies {
        sum += l
    }

    avg := sum / time.Duration(len(latencies))

    fmt.Println(avg.Microseconds()) // imprime só micro-segundos
}
```

Passo 2: Serviços do Sistema(Código da Implementação simplificada)

producer.go 1, M

simplified > producer.go > main

```go
func main() {
    addr := flag.String("addr",
        "localhost:9000",
        "queue server <host:port>")
    n := flag.Int("n", 1000, "number of messages to publish")
    flag.Parse()

    conn, err := net.Dial("tcp", *addr)
    if err != nil {
        log.Fatalf("dial: %v", err)
    }
    defer conn.Close()
    writer := bufio.NewWriter(conn)
    // to consume the "OK" acknowledgements      You, 8 minutes ago • U
    reader := bufio.NewReader(conn)

    for i := 0; i < *n; i++ {
        ts := time.Now().UnixNano()
        fmt.Fprintf(writer, "PUSH %d\n", ts)
        writer.Flush()
        // wait for ack so we don't overload TCP buffers (optional)
        reader.ReadString('\n')
    }
    log.Printf("published %d messages\n", *n)
}
```

Passo 2: Serviços do Sistema(Código da Implementação simplificada)

server.go 1

simplified > server.go > (*QueueServer).handleConn

```go
// QueueServer is a single-instance, in-memory FIFO queue.
Kennedy Melo, 5 days ago | 1 author (Kennedy Melo)
type QueueServer struct {
    mu    sync.Mutex
    queue []string
}

func (qs *QueueServer) handleConn(conn net.Conn) {
    defer conn.Close()
    reader := bufio.NewReader(conn)
    for {      Kennedy Melo, 5 days ago • init commit …
        line, err := reader.ReadString('\n')
        if err != nil {
            return // connection closed or error
        }
        line = strings.TrimSpace(line)

        switch {
        case strings.HasPrefix(line, "PUSH "):
            // PUSH <message>
            msg := strings.TrimPrefix(line, "PUSH ")
            qs.mu.Lock()
            qs.queue = append(qs.queue, msg)
            qs.mu.Unlock()
            conn.Write([]byte("OK\n"))

        case line == "PULL":
            qs.mu.Lock()
            if len(qs.queue) == 0 {
                qs.mu.Unlock()
                conn.Write([]byte("EMPTY\n"))
                continue
            }
            msg := qs.queue[0]
            qs.queue = qs.queue[1:]
            qs.mu.Unlock()
            conn.Write([]byte("MSG " + msg + "\n"))

        default:
            conn.Write([]byte("ERR unknown command\n"))
        }
    }
}
```

Passo 2: Serviços do Sistema(Código da Implementação simplificada)

Passo 2: Serviços do Sistema(Código da Implementação simplificada)

Passo 2: Serviços do Sistema

- Para a implementação RabbitMQ:
  - Estabelecimento de conexões com o broker RabbitMQ
  - Criação e gerenciamento de canais de comunicação
  - Declaração de filas para troca de mensagens
  - Publicação de mensagens pelos produtores
  - Consumo de mensagens pelos consumidores
  - Reconhecimento automático do recebimento de mensagens
- Para a implementação utilizando RabbitMQ utilizamos uma imagem docker no computador.

```
kennedy@kennedy-Inspiron-5480:~/Documentos/ufpe/2025.1/concorrentes/Go/atividades/atividade4/rabbitmq$ docker ps
CONTAINER ID   IMAGE                       COMMAND                  CREATED        STATUS          PORTS
                                     NAMES
b2e6741c1025   rabbitmq:3.13-management   "docker-entrypoint.s…"   3 days ago     Up 5 seconds    4369/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp, :::5672->5672/tcp, 15671/tcp, 15691-15692/tcp,
 25672/tcp, 0.0.0.0:15672->15672/tcp, :::15672->15672/tcp   rabbitmq
```

consumer_rmq.go 1, M

rabbitmq > consumer_rmq.go > ...

```go

func main() {
    n := flag.Int("n", 1000, "number of messages to expect before exiting")
    flag.Parse()

    conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
    if err != nil {
        log.Fatalf("dial: %v", err)
    }
    defer conn.Close()

    ch, err := conn.Channel()
    if err != nil {
        log.Fatalf("channel: %v", err)
    }
    defer ch.Close()

    q, err := ch.QueueDeclare("bench_queue", false, false, false, false, nil)
    if err != nil {
        log.Fatalf("queue: %v", err)
    }

    msgs, err := ch.Consume(q.Name, "", true, false, false, false, nil)
    if err != nil {
        log.Fatalf("consume: %v", err)
    }

    var latencies []time.Duration
    for msg := range msgs {
        sent, _ := strconv.ParseInt(string(msg.Body), 10, 64)
        lat := time.Now().UnixNano() - sent
        latencies = append(latencies, time.Duration(lat))
        if len(latencies) >= *n {
            break
        }
    }

    var sum time.Duration
    for _, l := range latencies {
        sum += l
    }

    avg := sum / time.Duration(len(latencies))

    fmt.Println(avg.Microseconds()) // imprime só micro-segundos
}
```

Passo 2: Serviços do Sistema (Código da implementação RabbitMQ)

producer_rmq.go 1

rabbitmq > producer_rmq.go > ...

```go
func main() {
    n := flag.Int("n", 1000, "number of messages to publish")
    flag.Parse()

    conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
    if err != nil {
        log.Fatalf("dial: %v", err)
    }
    defer conn.Close()

    ch, err := conn.Channel()
    if err != nil {
        log.Fatalf("channel: %v", err)
    }
    defer ch.Close()

    q, err := ch.QueueDeclare("bench_queue", false, false, false, false, nil)
    if err != nil {
        log.Fatalf("queue: %v", err)
    }

    for i := 0; i < *n; i++ {
        ts := time.Now().UnixNano()
        body := []byte(fmt.Sprintf("%d", ts))
        err = ch.Publish("", q.Name, false, false, amqp.Publishing{Body: body})
        if err != nil {
            log.Fatalf("publish: %v", err)
        }
    }
    log.Printf("published %d messages to RabbitMQ\n", *n)
}
```

Passo 2: Serviços do Sistema (Código da implementação RabbitMQ)

Passo 2: Serviços do Sistema

- Resultados
  - Mensagens entregues corretamente do produtor ao consumidor
  - Média de latência coletadas
  - Comparação de desempenho entre a implementação simplificada e o RabbitMQ
  - Preservação da ordem FIFO das mensagens durante todo o processo de transmissão

Passo 3

Métricas de Desempenho

Passo 3: Métricas de Desempenho

- Foi utilizada a Latência média entre o envio e recebimento de todas mensagens.

Passo 4

Parâmetros do Sistema

Passo 4: Parâmetros do Sistema

Parâmetro do Sistema	Valor
Hardware	Dell, Memória 15,3 GiB RAM, Intel® Core™ i7-8565U CPU @ 1.80GHz × 8 , Intel® UHD Graphics 620 (WHL GT2), GNOME 3.82.2 64 bits, disco 120,1 GB SSD.
Sistema Operacional	Ubuntu 18.04.6 LTS
Linguagem de programação	Go
Interfaces de rede	Ligadas
Fonte de Alimentação	Rede Elétrica
Processos em execução	Processos normais de rotina do sistema e navegador para chamada de vídeo

Passo 5

Escolha dos Fatores

Passo 5: Escolha dos Fatores

Fator	Nível
Algorítmo	*RabbitMq* e *Simplified*
Mensagens	10.000 e 200.000

Passo 6

Técnica de Avaliação

Passo 6: Técnica de Avaliação

Medição.

Passo 7

Parâmetro de carga de trabalho

Passo 7: Parâmetro de carga de trabalho

Parâmetro de Carga de Trabalho	*Valor*
Tamanho da Amostra	10.000 e 200.000
Número de invocações	30*10.000 e 30*200.000

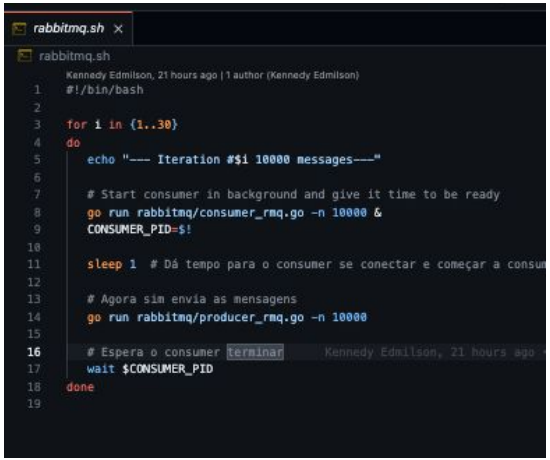
Passo 8

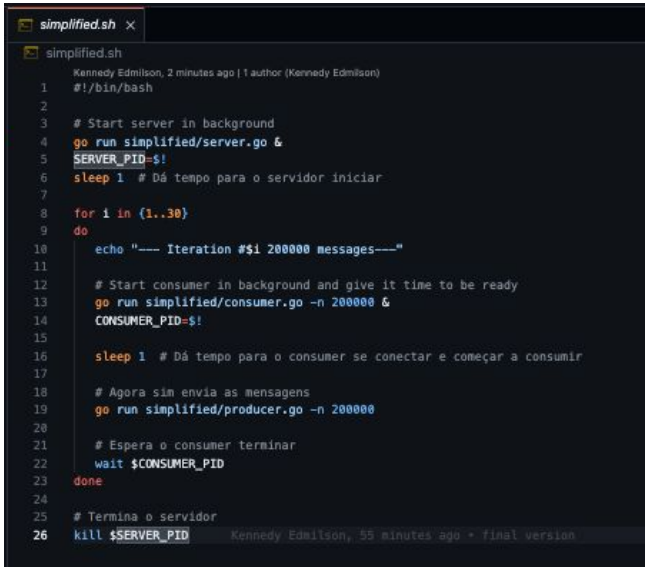
Projetar o experimento

Passo 8: Projetar o Experimento

O experimento foi projetado para comparar o desempenho de duas implementações de fila: uma FIFO simplificada sobre TCP e outra utilizando RabbitMQ. A principal métrica de avaliação é a latência média por mensagem, medida em microssegundos.

Para garantir a robustez dos dados, a metodologia consiste em executar 30 rodadas de testes para cada algoritmo (tanto para o Simplified quanto para o RabbitMQ) com 10.000 mensagens e depois com 200.000. Todo o processo é automatizado com scripts Bash, que orquestram as execuções e calculam a média global das latências ao final das 30 rodadas.



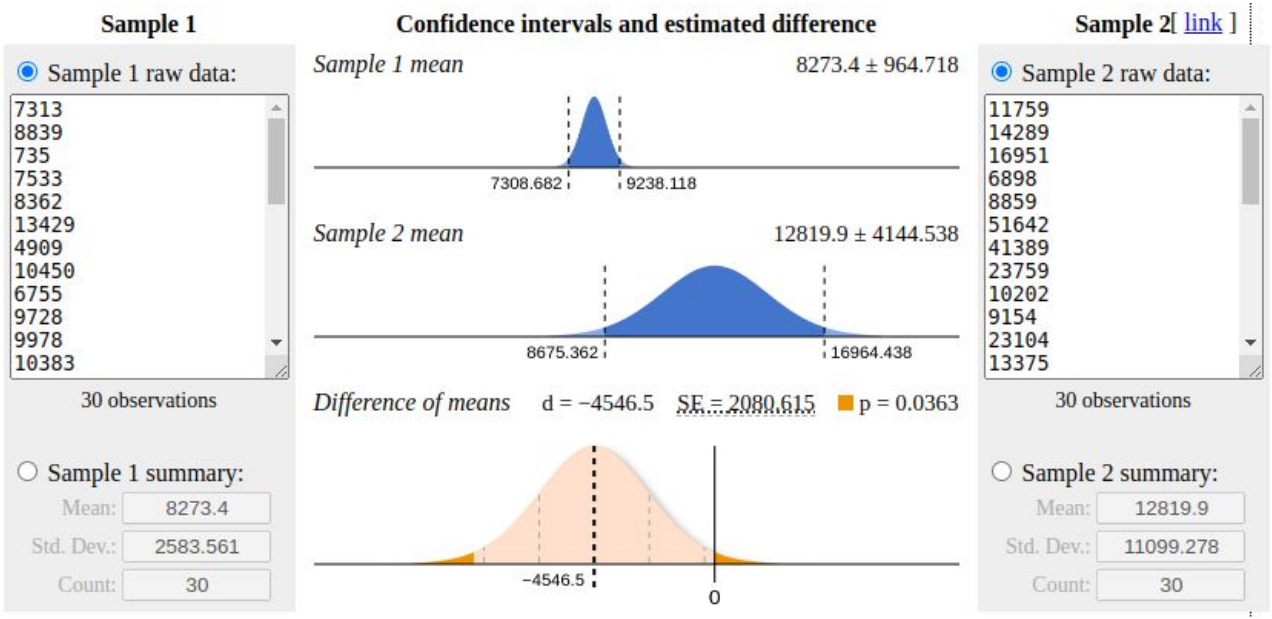


Passo 9

Análise e interpretação de resultados

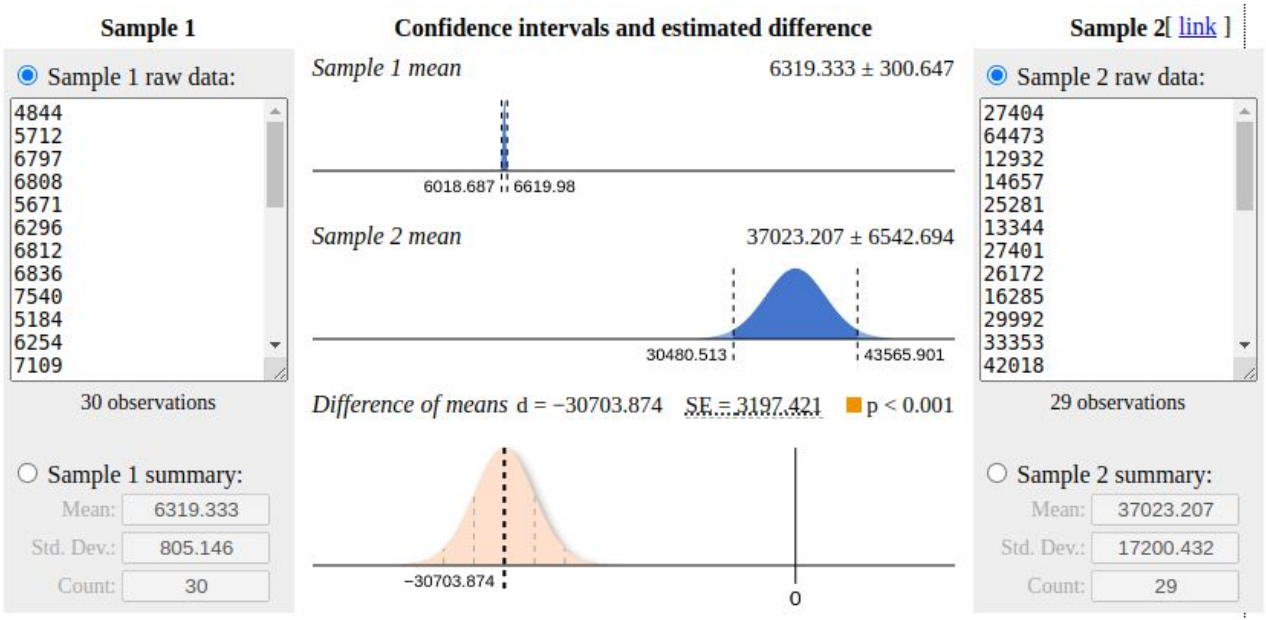
Passo 9: Analisar e Interpretar os Resultados (RabbitMQ x Simplified)

RabbitMQ/mensagens= 10.000 x Simplified/mensagens=10.000



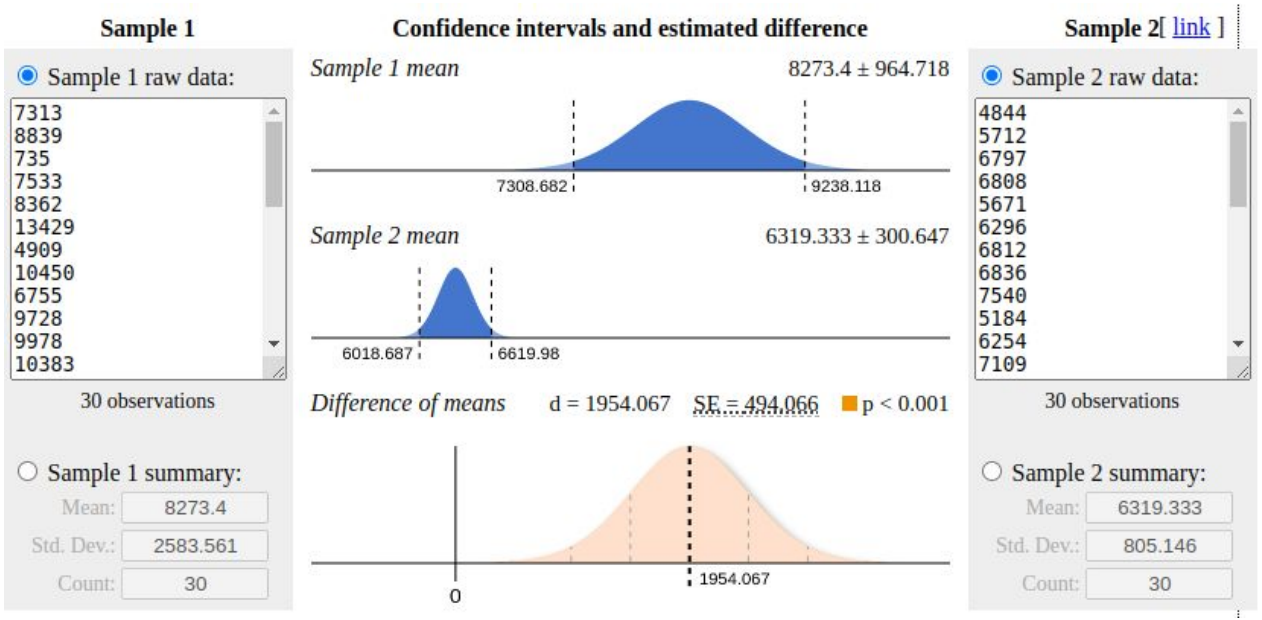
Passo 9: Analisar e Interpretar os Resultados (RabbitMQ x Simplified)

RabbitMQ/mensagens= 200.000 x Simplified/mensagens= 200.000



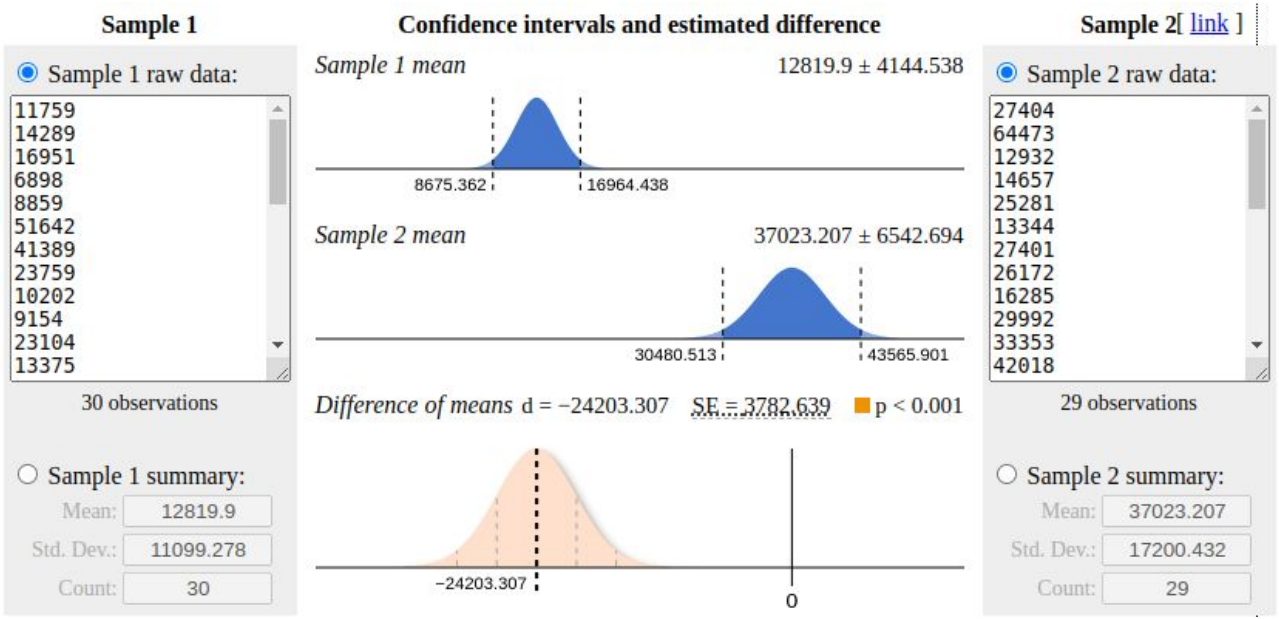
Passo 9: Analisar e Interpretar os Resultados (RabbitMQ x Simplified)

RabbitMQ/mensagens= 10.000 x RabbitMQ/mensagens=200.000



Passo 9: Analisar e Interpretar os Resultados (RabbitMQ x Simplified)

Simplified/mensagens= 10.000 x Simplified/mensagens=200.000



Passo 9: Apresentar a médias dos resultados.

Método de fila	Quantidade de mensagens (n = 10.000)	Quantidade de mensagens (n = 200.000)
Simplified	12819,90 µs	37023,21 µs
RabbitMQ	8506,73 µs	6319,33 µs

Passo 9: Analisar e Interpretar os Resultados (RabbitMQ x Simplified)

Métrica	n = 10 000	n = 200 000
Latência Simplified (µs)	12 819,9	37 023,2
Latência RabbitMQ (µs)	8 506,7	6 319,3
Speed-up RabbitMQ / Simplified	**1,51 ×**	**5,86 ×**
Throughput Simplified (msg/s)	78	27
Throughput RabbitMQ (msg/s)	118	158

Passo 9: Apresentar o desvio padrão dos resultados.

Método de fila	Quantidade de mensagens (n = 10.000)	Quantidade de mensagens (n = 200.000)
Simplified	11099,28 µs	17200,43 µs
RabbitMQ	2160,76 µs	805,15 µs

Passo 9: Analisar e Interpretar os Resultados (RabbitMQ x Simplified)

Os resultados apontam uma superioridade clara do RabbitMQ, sobretudo à medida que o volume de mensagens cresce. Com 10.000 envios, a fila simplificada apresentou uma latência média de 12,8 ms, enquanto o RabbitMQ concluiu o mesmo teste em 8,5 ms. Quando a carga aumentou para 200.000 mensagens, a disparidade tornou-se ainda mais evidente: o RabbitMQ reduziu sua latência para 6,3 ms, ao passo que a implementação simplificada disparou para 37 ms.

Essa diferença decorre, essencialmente, de como cada solução lida com concorrência e comunicação. O RabbitMQ opera sobre o protocolo AMQP, que já contempla pré-busca e envio em lote: várias mensagens podem ser entregues num único ciclo de rede, diluindo o custo fixo de cada ida-e-volta. Além disso, o broker, escrito em Erlang, distribui o trabalho por múltiplos processos leves, minimizando contenção de locks e aproveitando plenamente os núcleos disponíveis.

A versão simples, por outro lado, segue um modelo estritamente síncrono: cada PULL exige um novo pedido TCP, protegido por um mutex global que coordena produtores e consumidores. Quando a fila cresce, esse mutex torna-se um gargalo e as operações de slice na memória fazem o coletor de lixo trabalhar mais, elevando a latência.

Em suma, o RabbitMQ converte o aumento de carga em ganho de eficiência graças ao pipeline assíncrono, ao controle de fluxo e à sua arquitetura paralela; a solução caseira, embora válida para demonstrações ou pequenos testes, não possui esses mecanismos e, por isso, escala mal quando submetida a volumes maiores de mensagens.

Link para o Repositório - Github

https://github.com/KennedyMelo/atividade4-distribuidos