EP1

Lucas Seiki Oshiro

Implementação

Estruturas de dados

Filas e trie

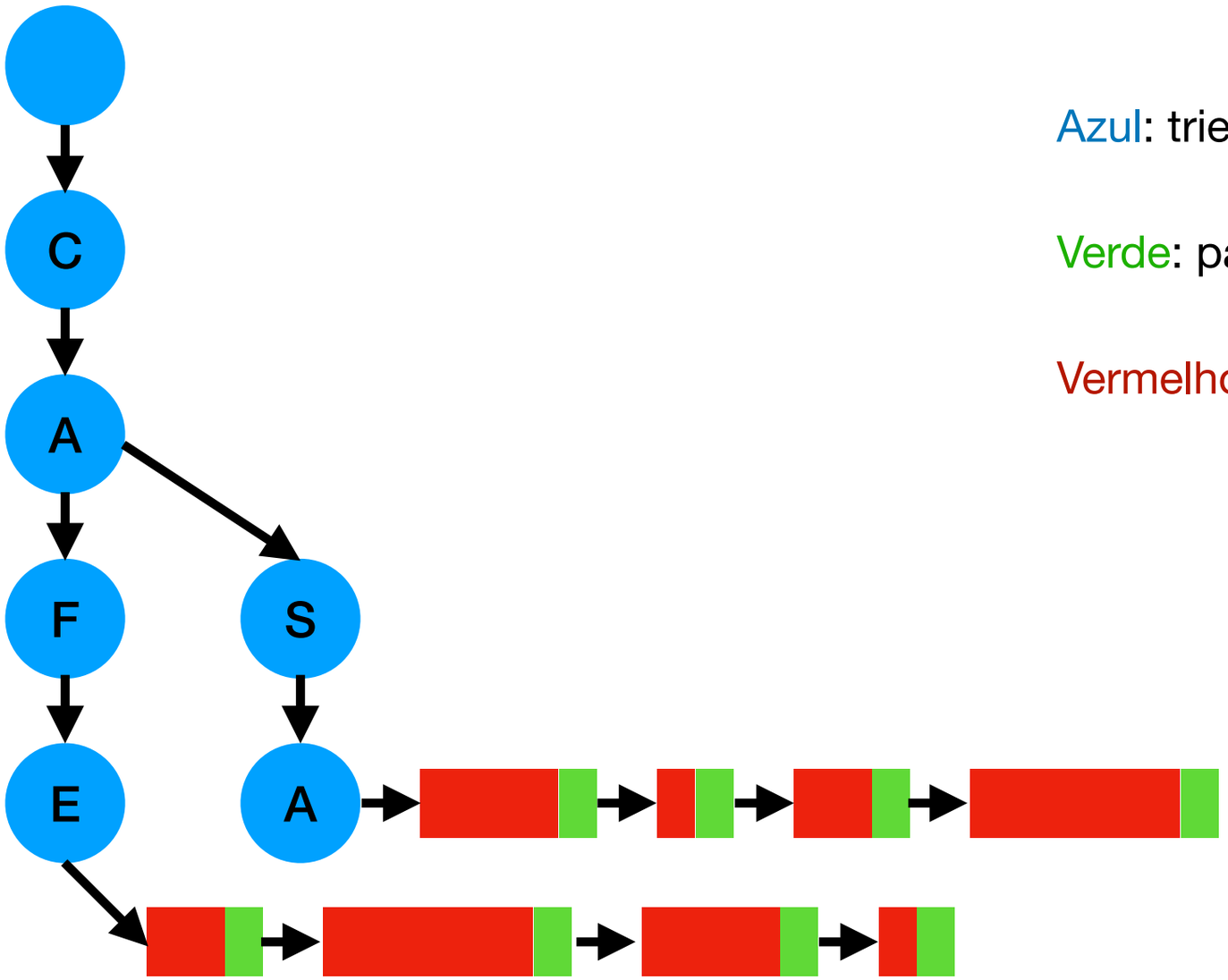
- Filas são **listas encadeadas**
- Células das filas são **arrays flexíveis**
- Mapeamento nome -> fila é uma **trie**

```c
typedef struct q_node {
    struct q_node *parent;
    int length;
    char body[1];
} q_node;
```

```c
q_node *n;
int length = strlen(body);

n = malloc(sizeof(*n) + length * sizeof(char));
n->parent = NULL;
n->length = length;
strcpy(n->body, body);
```

Estruturas de dados

Filas e trie



Azul: trie

Verde: parte de tamanho fixo da célula

Vermelho: parte de tamanho flexível da célula

Estruturas de dados

Mensagens

- Podemos copiar os bytes da entrada direto para as **structs**
- É necessário tratar **endianess!**
- __attribute__((packed)) para evitar **padding!**

```c
typedef struct {
    uint8_t msg_type;
    uint16_t channel;
    uint32_t length;
} __attribute__((packed)) amqp_message_header;
```

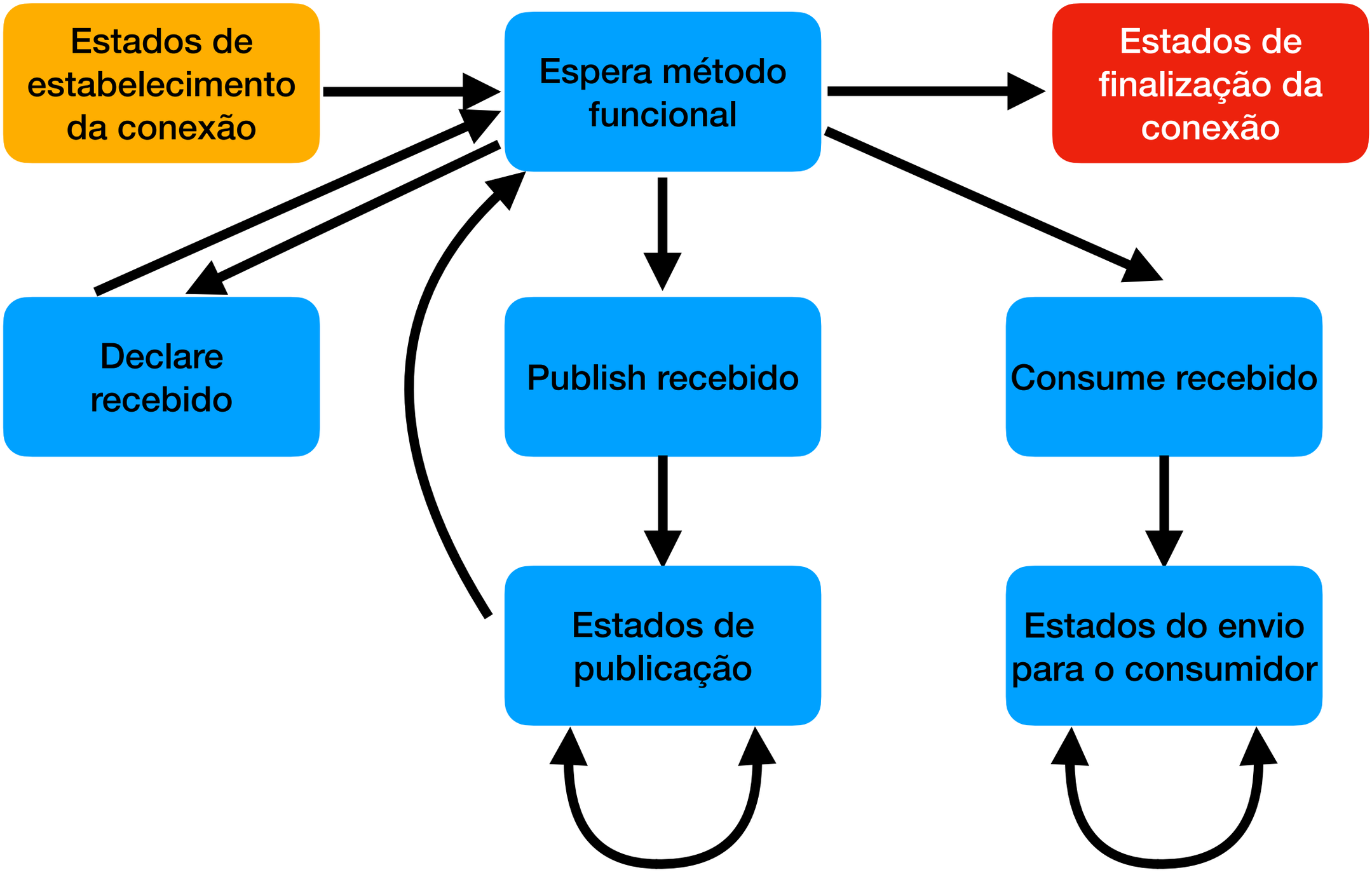
```c
static int parse_message_header(
    char *s,
    size_t n,
    amqp_message_header *header
    ) {
    size_t header_size = sizeof(amqp_message_header);

    if (n < header_size) return 1;
    memcpy(header, s, header_size);

    header->channel = ntohs(header->channel);
    header->length = ntohl(header->length);
    return 0;
}
```

Comunicação
Conexões e threads

- Cada conexão em uma **thread**
- Filas ficam **compartilhadas** entre threads
- Cada thread tem um **estado**
- Os estados formam uma **máquina de estados**
- **Envio** e **recebimento** de mensagens mudam os estados
- **Manipulação das filas** também mudam os estados

Comunicação

Máquina de estados (resumida)



Comandos

Declare e Publish

- **Declare:**
    - Cada declaração de fila apenas declara a fila na trie
    - Declarações com nome já existente: **não faz nada**
- **Publish:**
    - O publish fica em loop
    - A entrada no loop: mensagem **publish** e **content header**
    - No loop: conteúdos são recebidos **indefinidamente**

Comandos

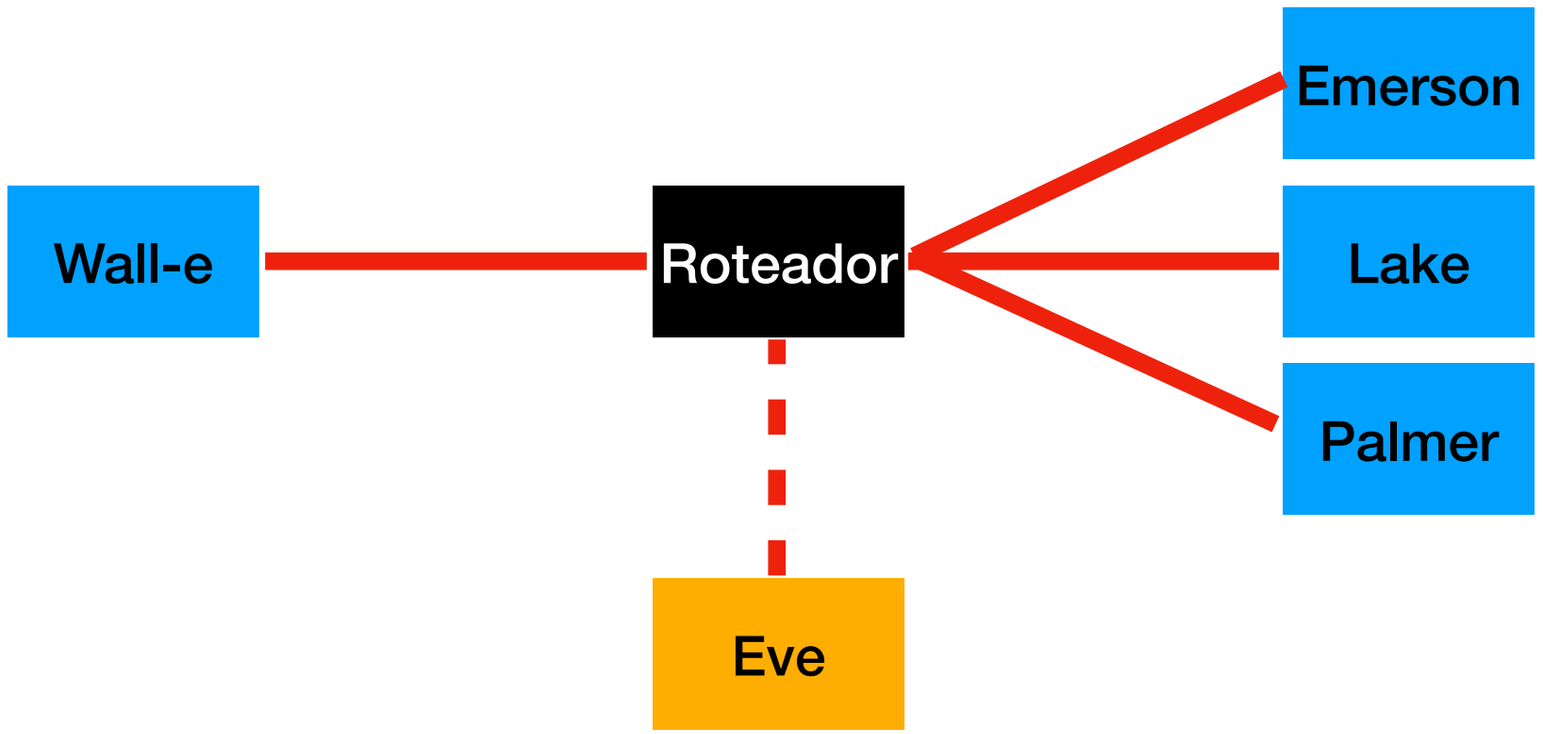
Consume

- O consume também fica em loop
- A entrada no loop: mensagem **consume**
- No loop:
    - espera-se a vez da thread em **round-robin**
    - espera-se até que haja algum valor na fila
    - **retira-se** um valor da lista
    - **avisa-se** a próxima thread que é a vez dela
    - **envia-se** o valor para o cliente

Experimentos

Ambiente

Rede e máquinas



Máquina	CPU	RAM	OS	Conexão
Wall-e	Intel i5 x 2 (x86 64)	8GB	Manjaro Linux	Cabeada
Eve	Apple M1 x 8 (ARM 64)	8GB	Mac OS Monterey	Wireless
Emerson	ARM Cortex A8 x 1 (ARM 32)	230MB	Debian 10 Buster	Cabeada
Lake	ARM Cortex A8 x 1 (ARM 32)	484MB	Debian 10 Buster	Cabeada
Palmer	BCM 2835 x 1 (ARM 32)	432MB	Debian 11 Bullseye	Cabeada

Ambiente

Rede

- Roteador: Linksys WRT54G v8
    - Firmware dd-wrt
- Rede **sem acesso à internet**
- Throughput (medido com iperf)
    - Emerson -> Wall-e: **94.3 Mbits/s**
    - Lake -> Wall-e: **94.4 Mbits/s**
    - Palmer -> Wall-e: **53.7 Mbits/s**
- **Túnel SSH nas conexões!**
    - Necessário para o RabbitMQ quando sem autenticação
    - Não é necessário para o EP1, mas foi usado para manter o mesmo ambiente

Ambiente

Papel das máquinas

- Eve: **acesso remoto**, não participa ativamente
- Wall-e: **servidor**
- Emerson, Lake: **publishers**
- Palmer: **consumer**

Metodologia

- **Controle**: Wall-e em repouso, apenas executando o Manjaro
- **Experimento 1**: Wall-e executando o EP1
- **Experimento 2**: Wall-e executando o RabbitMQ

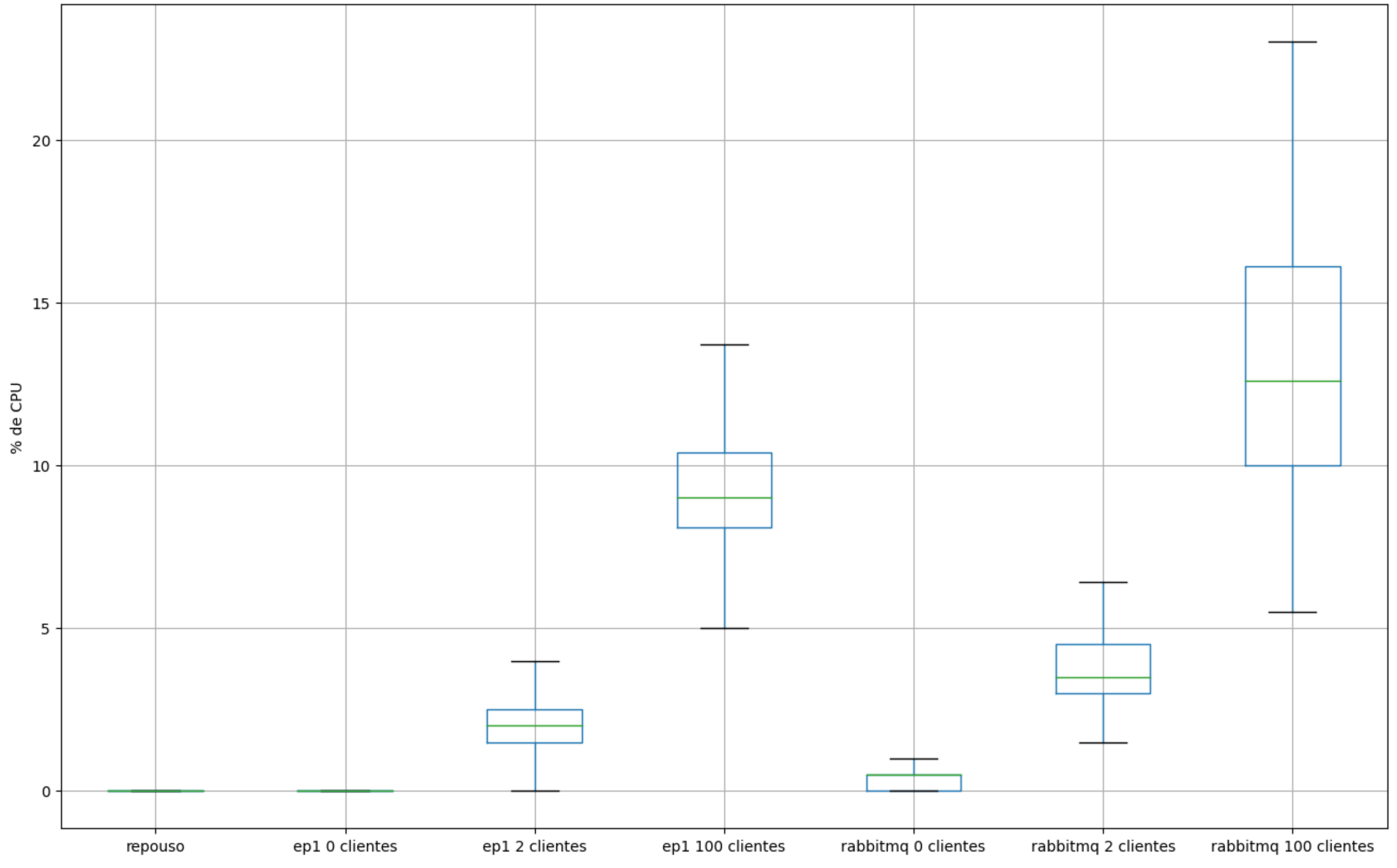
- Logs (usando raspberry-log)
  - **Wall-e**: CPU, RAM e rede
  - **Emerson**, **Lake** e **Palmer**: rede

Análise

- Observação do **controle**: para saber qual o comportamento normal e qual o impacto do EP1
- Observação do **rabbitmq**: comparação com um sistema já existente
- Observação das máquinas **clientes**: para saber onde foi maior o consumo

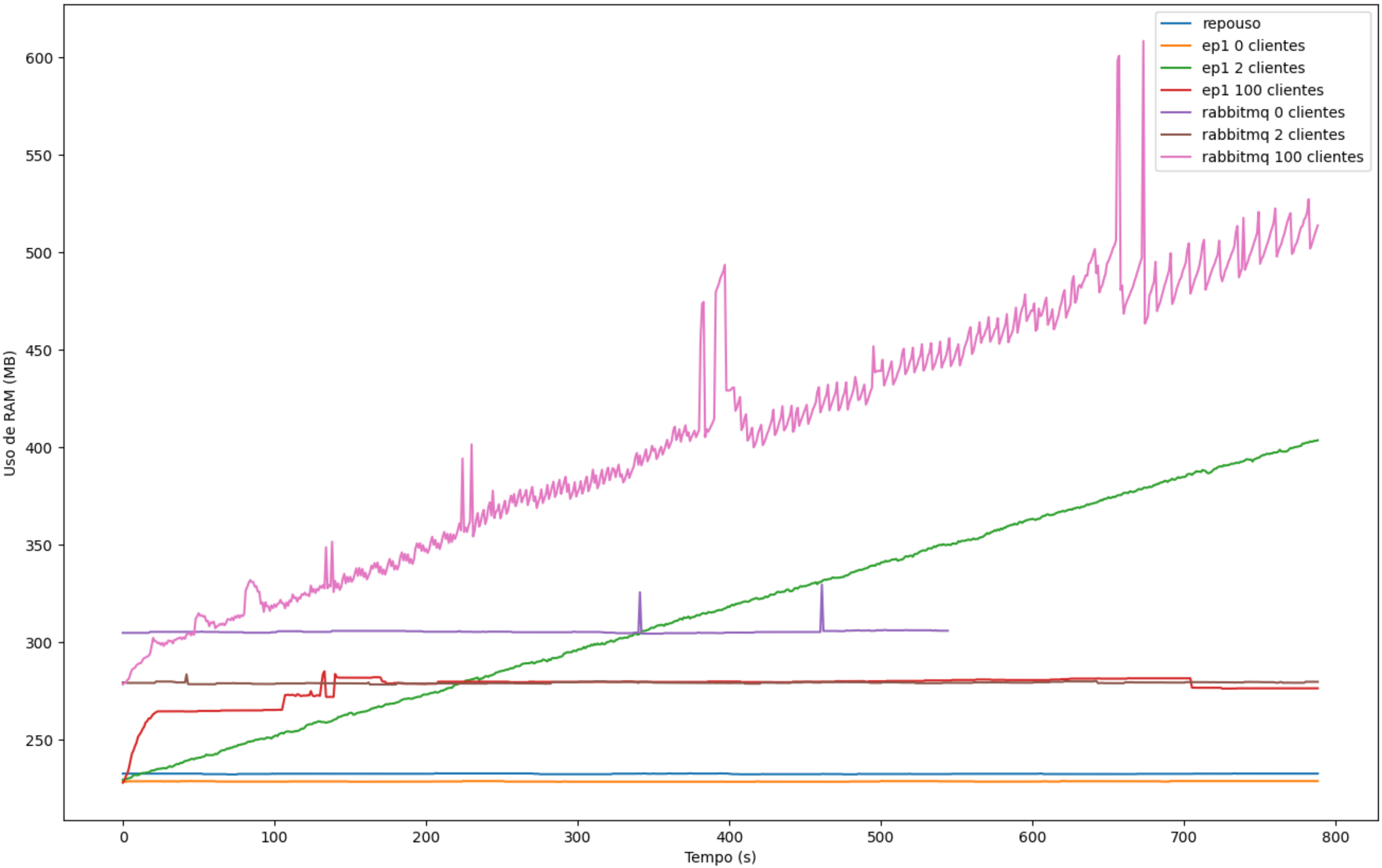
Observações

Wall-e (servidor): % de CPU



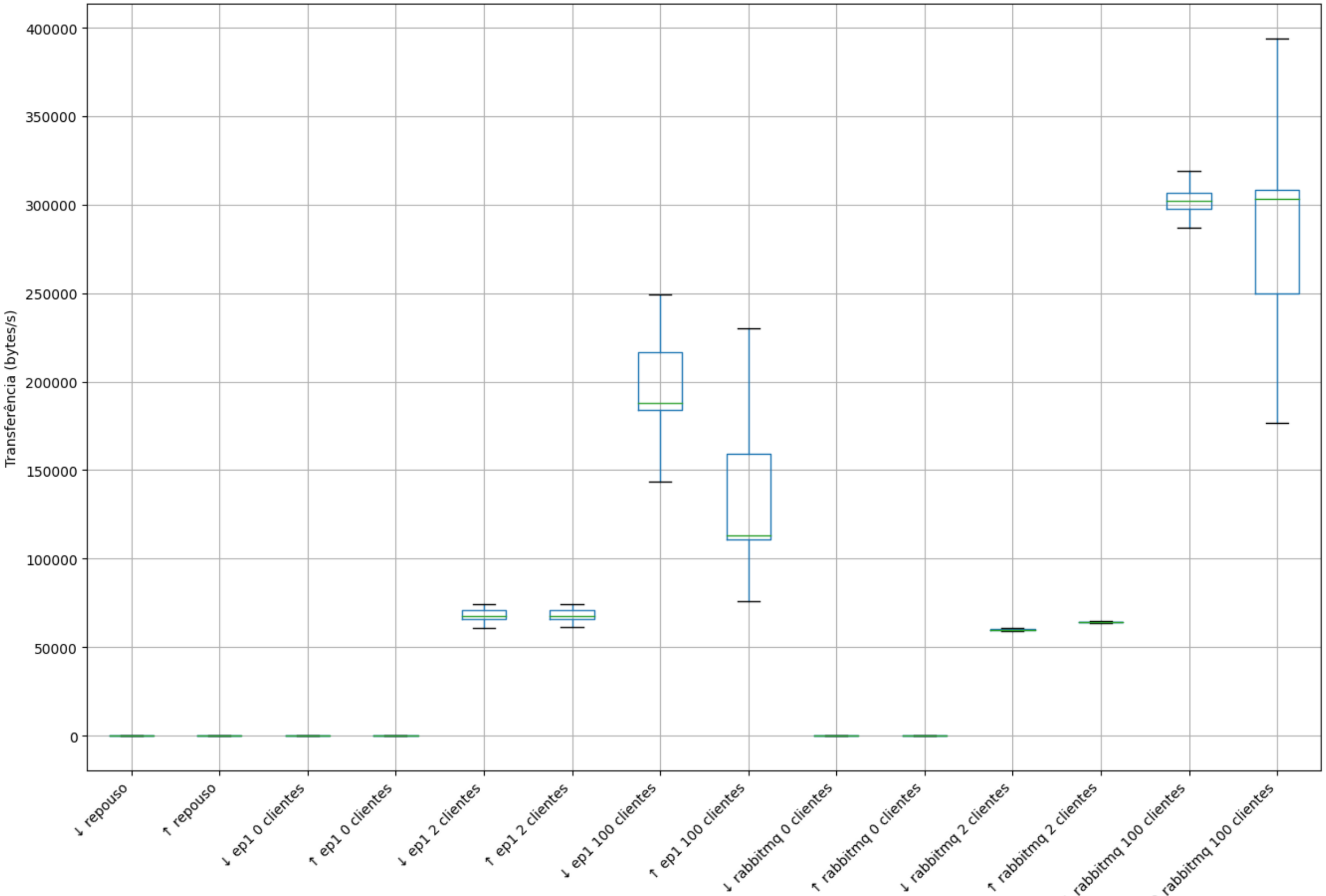
Observações

Wall-e (servidor): Memória



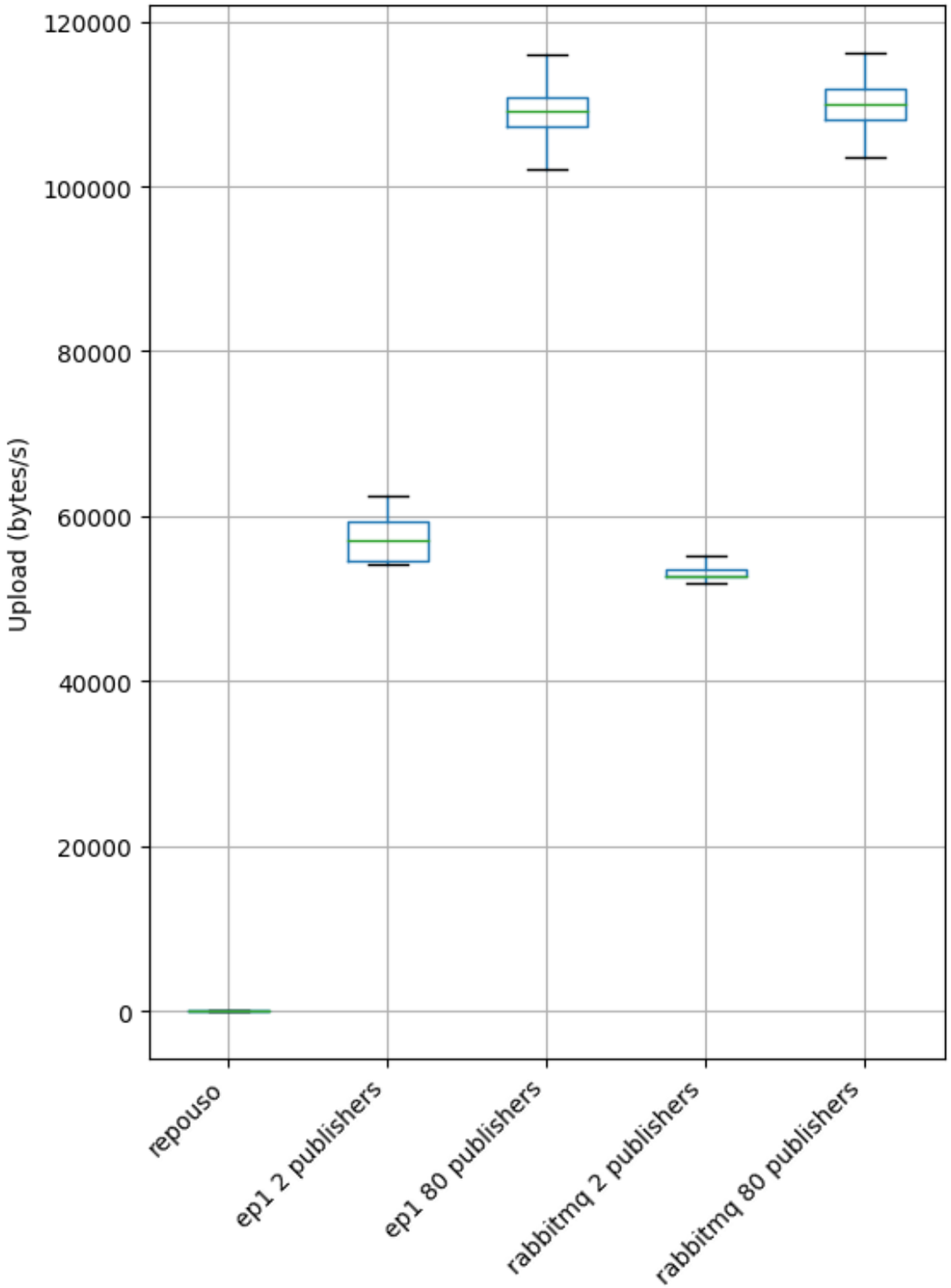
Observações

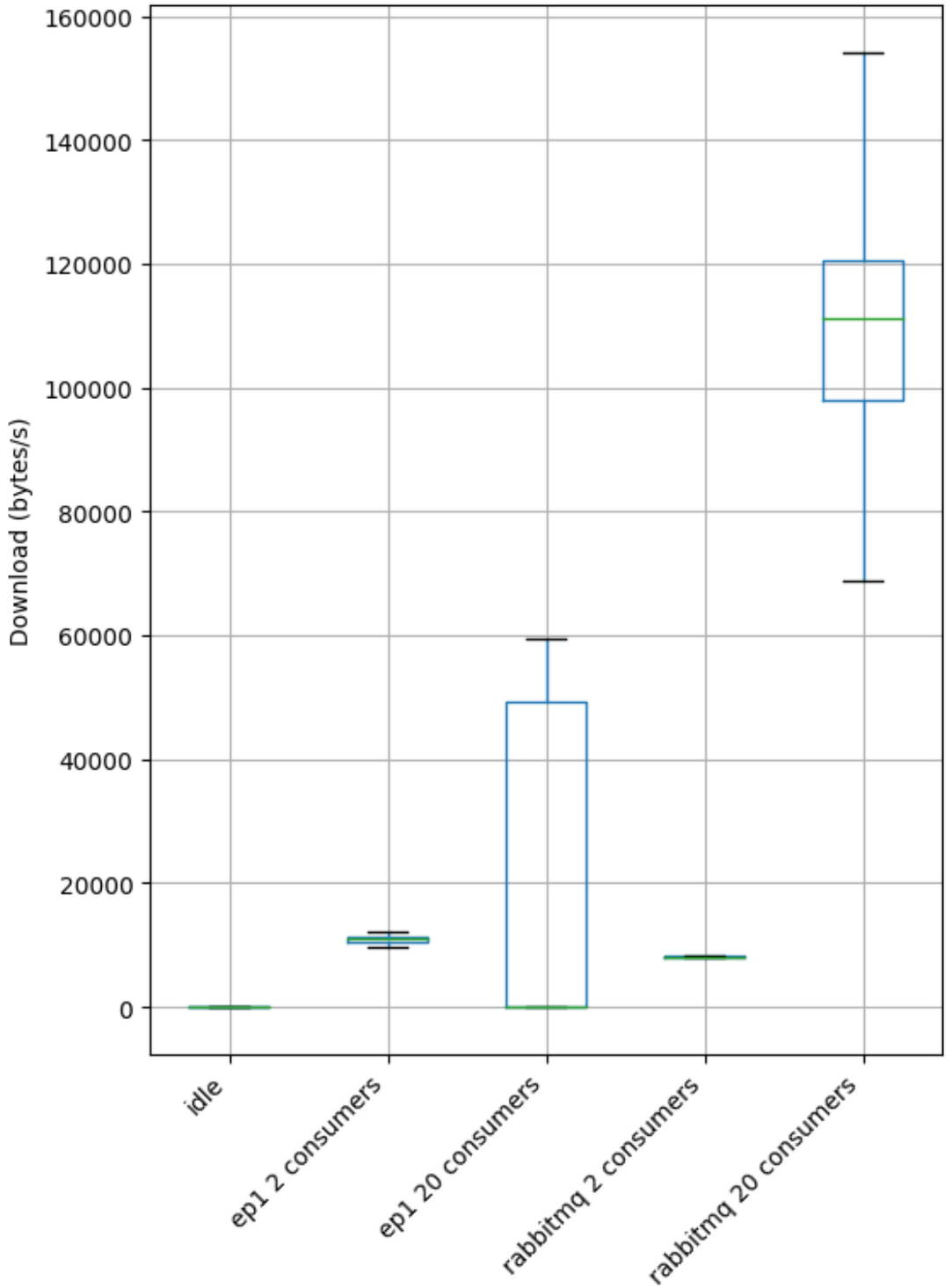
Wall-e (servidor): rede



Observações

Lake (publisher) e Palmer (consumer)





Conclusão

- O EP1 ocupou **menos** CPU e memória que o RabbitMQ
    - Possível causa: o EP1 é bem mais simples e é executado nativamente. O RabbitMQ é executado na máquina virtual de Erlang
- Com 1 producer e 1 consumer, a memória do EP1 cresceu **linearmente**.
    - Possível causa: chegou mais mensagens do que o consumer conseguiu receber
- Com 80 producers e 20 consumers, a memória do RabbitMQ também apresentou comportamento parecido
- O RabbitMQ apresentou **maiores** taxas de upload e download.

Obrigado!