Computação Concorrente (DCC/UFRJ)

Aula 11: Programação concorrente usando troca de mensagens

Prof. Silvana Rossetto

5 de junho de 2012

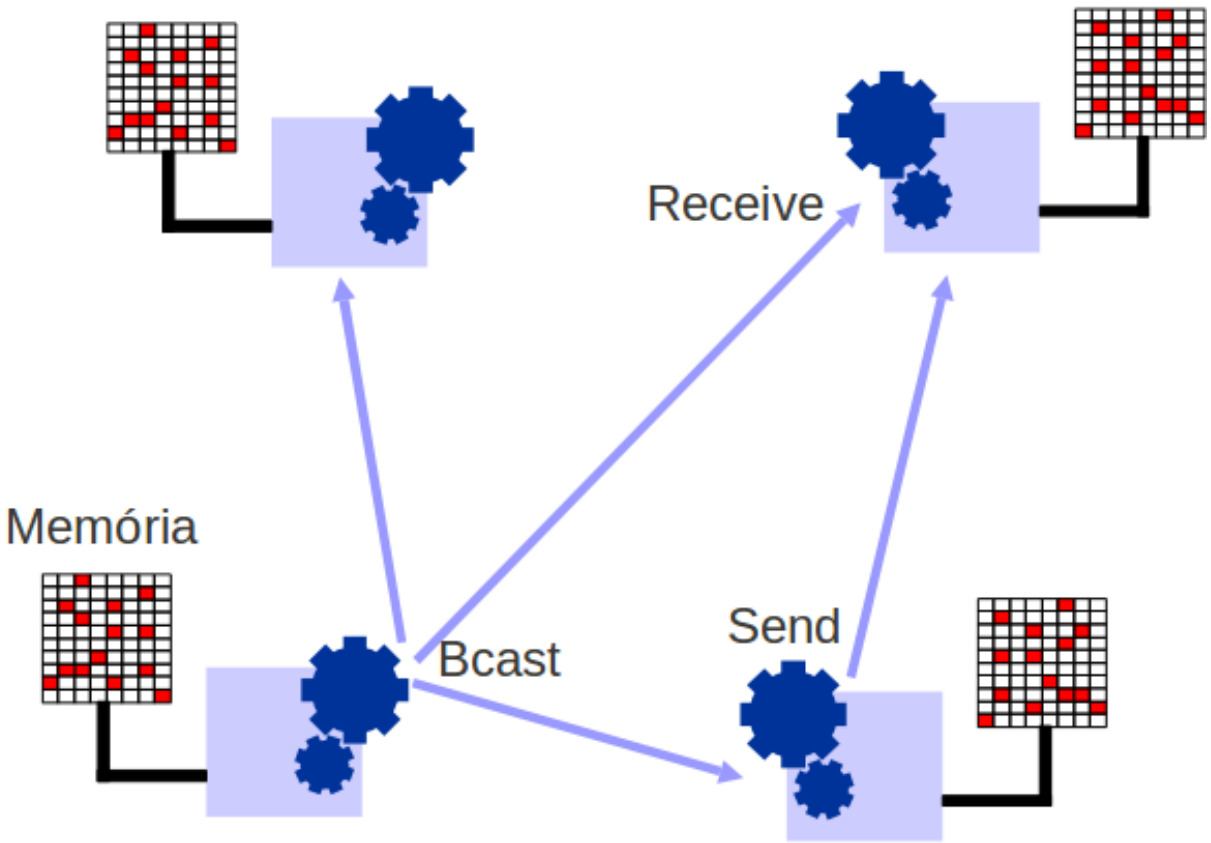
Modelos de programação para aplicações distribuídas

- **Aplicações distribuídas**: programas que executam em diferentes máquinas
- Importância de utilizar **abstrações de programação** que "facilitem" a vida do programador
  - idéia: oferecer **facilidades para troca de mensagens** ou estender modelos conhecidos da programação local
  - ex., memória compartilhada, chamada de procedimentos, orientação a objetos, eventos

Exemplos de bibliotecas de troca de mensagens

- **PVM**: biblioteca para troca de mensagens entre processos de uma aplicação
- **MPI**: definição de uma biblioteca de funções que permite diferentes implementações (ex. MPICH e MPILAM)
- **OpenMP**:

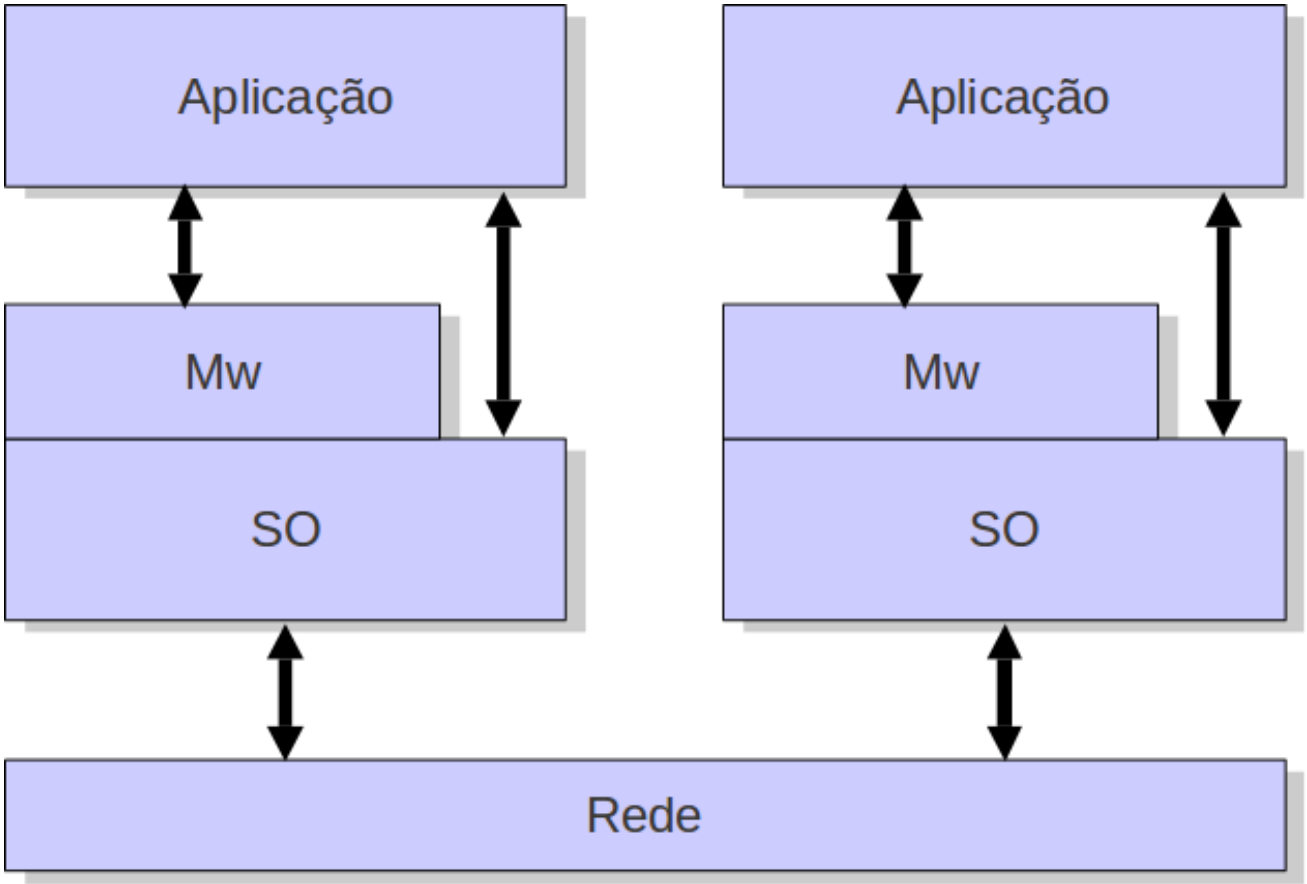
Primitivas adicionais para troca de mensagens



Middleware: conceitos básicos

- **Middleware** (Mw): camada de serviços adicionada entre o **sistema operacional de rede** e as **aplicações**
- **Funcionalidades da camada de middleware**:
  1. ocultar a heterogeneidade das arquiteturas de sistema (sw e hw)
  2. oferecer transparência de localização
  3. definir protocolos de comunicação (ex., *request/reply*) acima dos protocolos de transporte (TCP, UDP)
  4. permitir que as aplicações sejam escritas usando mais de uma linguagem de programação
  5. ...

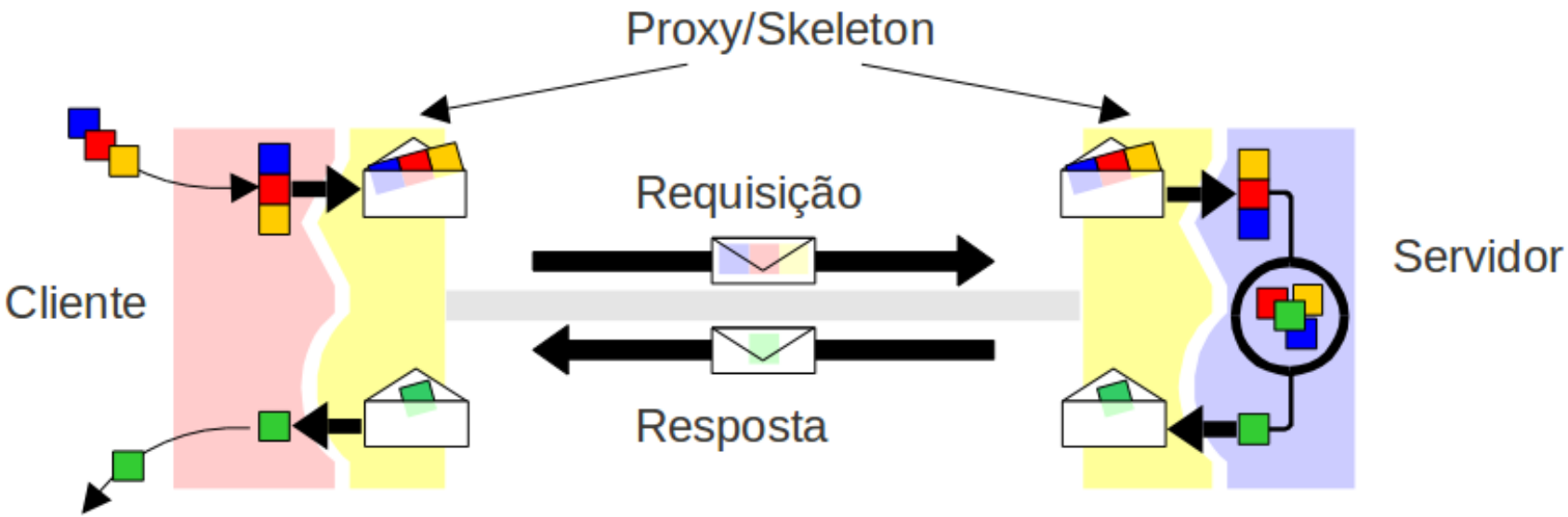
Camada de middleware



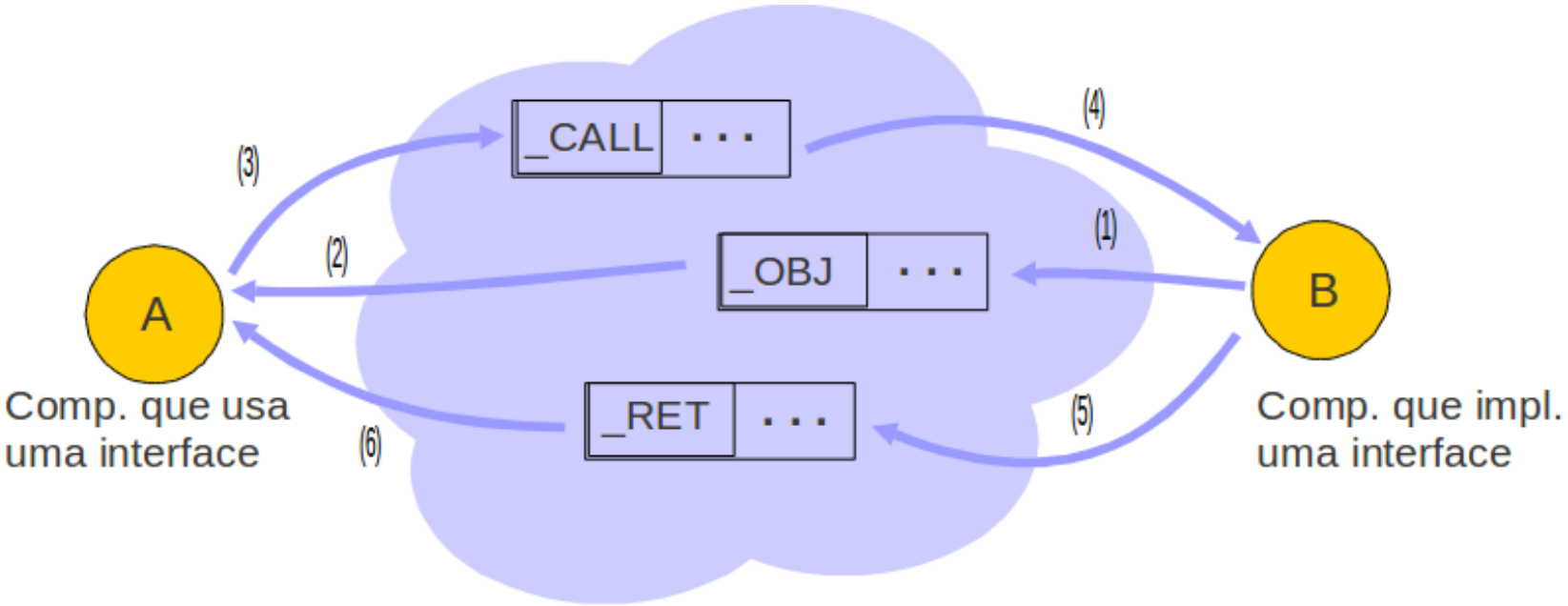
Abstrações de programação distribuída

1. **RPC**: extensão do modelo de chamada local de procedimento, particularmente adequado para o paradigma cliente/servidor
2. **RMI**: extensão do modelo de programação baseado em objetos, permite a comunicação entre objetos que executam em diferentes processos/máquinas
3. **Eventos**: extensão do modelo de interfaces gráficas (componentes que respondem a eventos), permite que objetos recebam notificações de eventos de outros objetos para os quais registram interesse

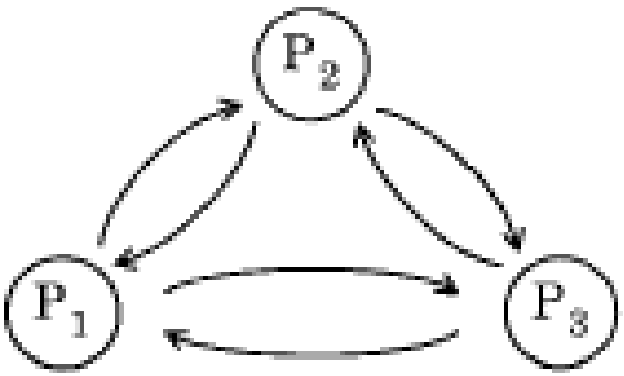
Abstrações de programação distribuída



Espaço de tuplas



Interação entre processos via troca de mensagens

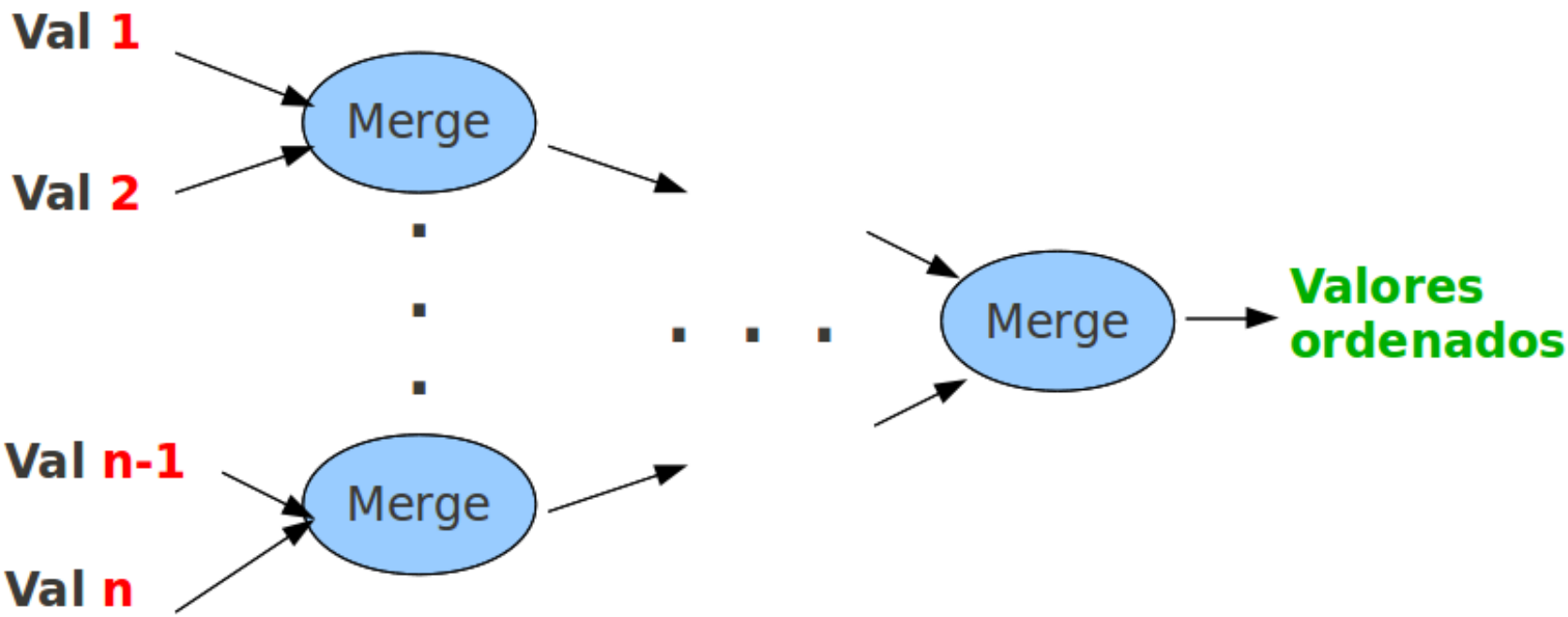


- Considerando as primitivas básicas de troca de mensagens **send** e **receive**, existem diferentes formas nas quais processos de uma aplicação concorrente podem interagir
- O ponto central é **entender as decisões relacionadas com a comunicação entre as partes**

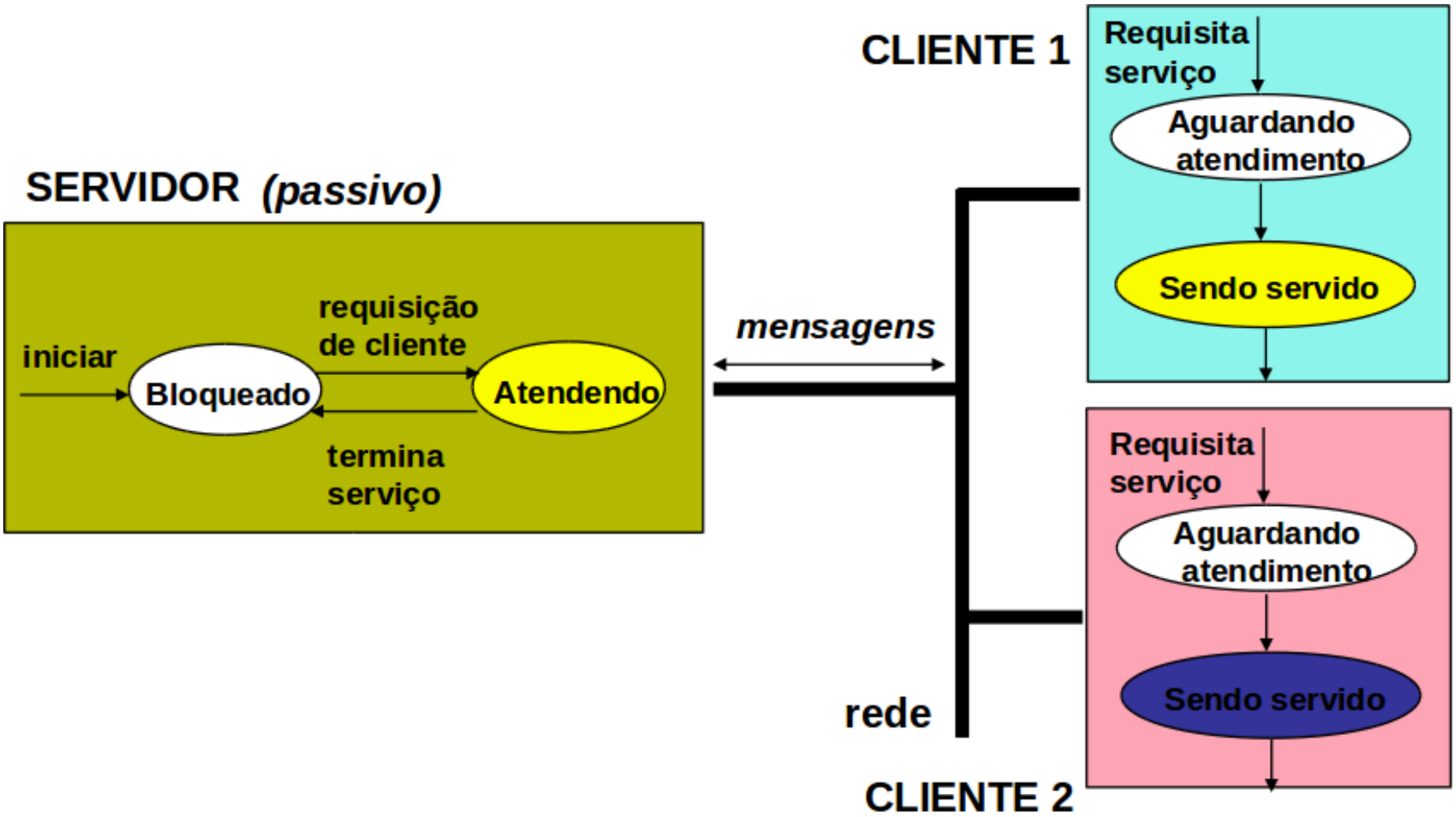
Modelos de interação entre processos

- **Filtros**: processos recebem mensagens de um ou mais canais de entrada e enviam mensagens para um ou mais canais de saída
- **Cliente/Servidor**: processos servidores manipulam requisições de processos clientes
- **Ponto-a-Ponto**: processos interagem aos pares de forma:
  - centralizada (todo processo comunica-se apenas com um processo central)
  - simétrica (todos os processos podem comunicar-se com todos os outros)
  - circular (cada processo comunica-se com um vizinho a esquerda e outro a direita, formando um círculo)

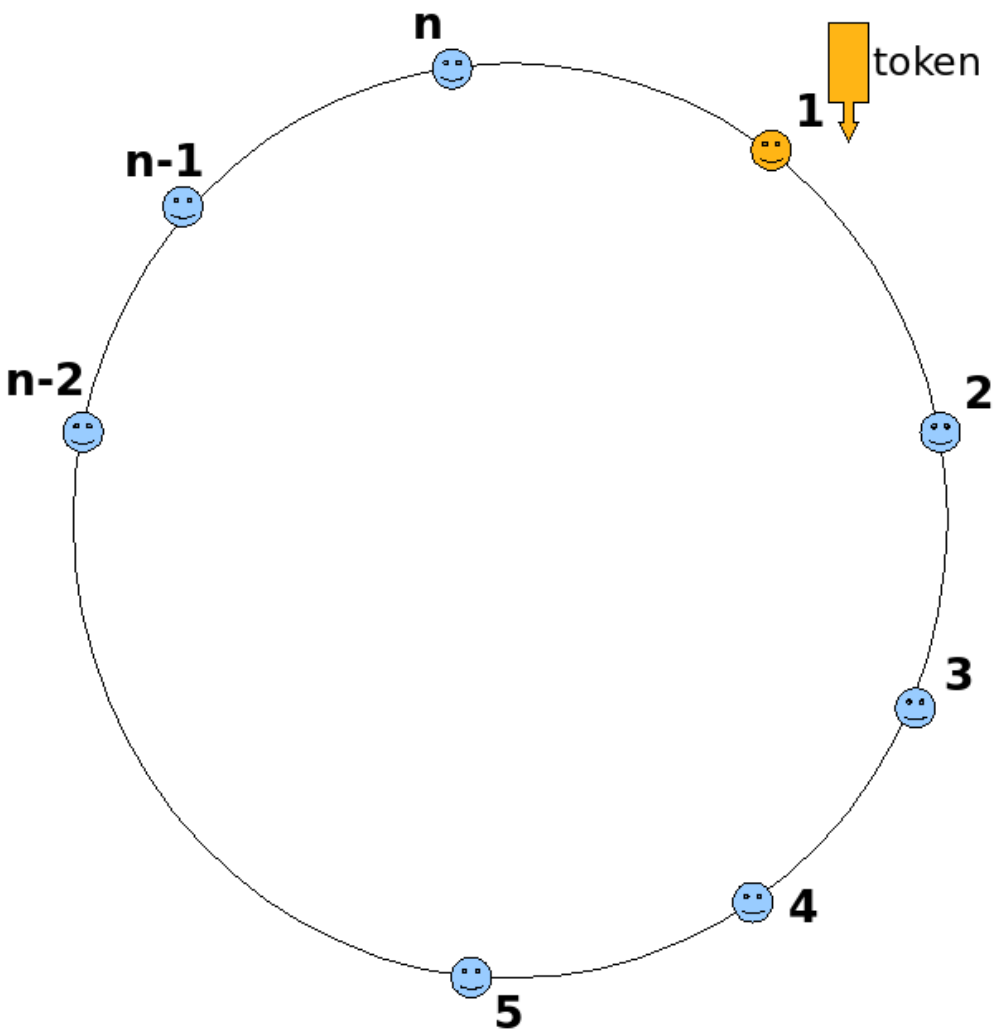
Exemplo filtro



Exemplo cliente/servidor



Exemplo ponto-a-ponto



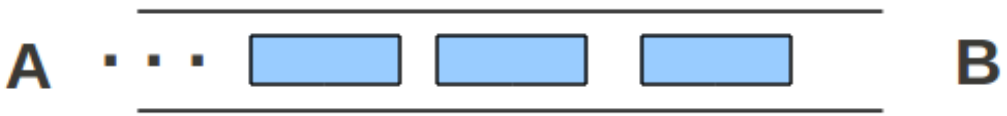
Exemplos de padrões de interação entre processos

- Fluxo de dados através de uma **rede de filtros**
- Requisições e respostas entre **clientes e servidores**
- **Passagem de bastão** entre vizinhos
- **Interações ida-e-volta** (*heartbeat*) entre vizinhos
- **Broadcast** entre processos
- **Trabalhadores replicados** e **bolsa de tarefas compartilhada**

Notação com troca de mensagem assíncrona

Notação para definição de canais

- Um canal é uma **fila de mensagens que foram enviadas e ainda não foram recebidas**:
- **chan** $ch(f_1 : t_1, ..., f_n : t_n)$, onde: ch é o nome do canal, f_i é o nome de um campo de dado (opcional) e t_i é o tipo do campo de dado
- ex., **chan** in(char); **chan** file (count: int, buffer:ptr[*]char)



Notação com troca de mensagem assíncrona

Notação para a primitiva send

- Um processo envia uma mensagem para o canal **ch** fazendo: **send** $ch(expr_1, ..., expr_n)$, onde $expr_i$ são expressões cujo resultados são do tipo correspondente ao campo do canal
- O efeito de executar **send** é avaliar as expressões e colocar a mensagem no final da fila do canal

Notação com troca de mensagem assíncrona

Notação para a primitiva receive

- Um processo recebe uma mensagem de um canal **ch** fazendo: **receive** $ch(var_1, ..., var_n)$
- O efeito de executar **receive** é esperar até que exista ao menos uma mensagem na fila do canal, a mensagem no início da fila é removida e os campos são associados às variáveis var_i correspondentes (**receive bloqueante, o processo receptor não fica em espera ocupada**)

Notação com troca de mensagem assíncrona

Outras operações sobre canais

- **empty(ch)**: usada para o processo não bloquear no canal, caso não exista mensagens disponíveis

Exemplo: exclusão mútua distribuída

Descrição do problema

- Trata-se do **problema de sincronização clássico** que visa garantir que **no máximo um processo de cada vez executa código que acessa um recurso compartilhado**
- Normalmente, a EMD é uma componente de problemas maiores
  - ex., consistência em sistemas de arquivos distribuídos ou sistemas de banco de dados distribuídos

Exclusão mútua distribuída com bastão circulante

- A tarefa é desenvolver **protocolos de entrada e saída**: os processos devem executar esses protocolos antes e depois de entrar na SC
  - devem garantir EM, evitar deadlock e espera desnecessária, e justiça
- A entrada na SC será controlada por meio de um **bastão circulante**

Um **bastão** (*token*) é um tipo especial de msg que pode ser usada para pedir permissão para executar uma ação ou para obter informação de estado

Exclusão mútua distribuída

Notação Andrews

- Um conjunto de processos **Helper[1:n]** (um para cada processo **P[1:n]**) formam um anel e compartilham um token
- A posse do token significa "permissão para entrar na SC"
- O anel é representado por um **vetor de canais** (o token é uma msg vazia)
- Os processos cooperam para garantir o seguinte predicado:

DMUTEX: ($\forall i : 1 \leq i \leq n : P[i]$ está na sua SC $\rightarrow$ Helper[i] tem o token) AND há exatamente um token

Algoritmo para exclusão mútua distribuída

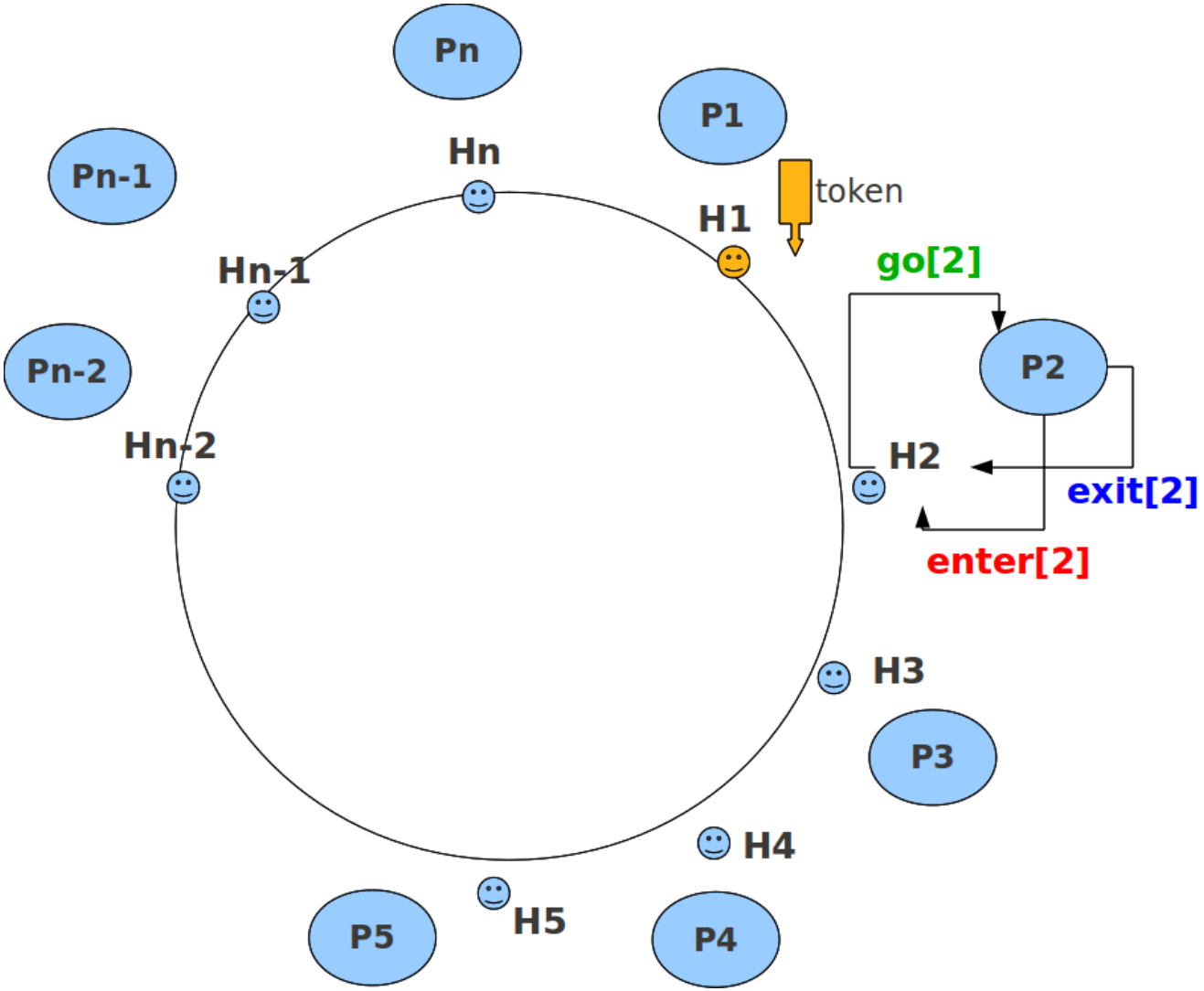
```
chan token[1:n](), enter[1:n](), go[1:n](), exit[1:n]();

[Helper[i:1..n]::]
while(true) {
   receive token[i](); //espera o token
   if not(empty(enter[i])) {
       //checa P[i] quer entrar na SC
      receive enter[i](); //extrai a msg do canal
      send go[i](); //envia a msg de permissão para SC
      receive exit[i]() //aguarda a msg de saída da SC
   }
   send token[i mod n+1](); //passa o token adiante
}
```

Algoritmo para exclusão mútua distribuída

```
[P[i:1..n]::]
while(true) {
   send enter[i](); //avisa que quer entrar na SC
   receive go[i](); //aguarda permissão para a SC
   //...executa a seção crítica (SC)
   send exit[i](); //avisa que terminou a SC
   //...executa fora da SC
}
```

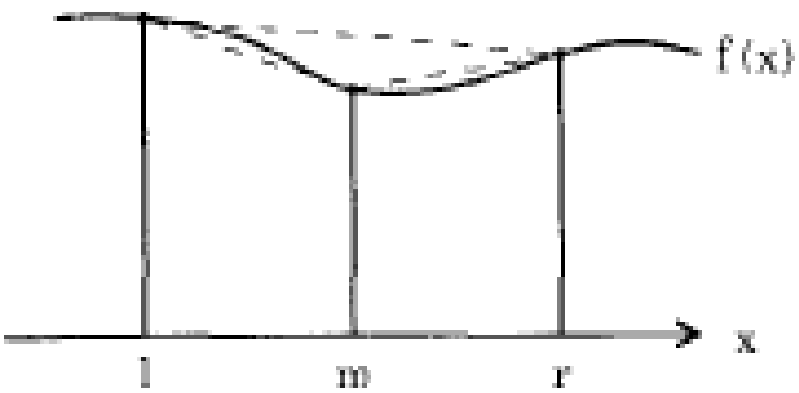
Algoritmo para exclusão mútua distribuída



Trabalhadores replicados e bolsa de tarefas

- Exemplo de situação onde há **replicação de código** (ao invés de replicação de dados)
- Veremos como exemplo uma **solução paralela para o problema de quadratura para integração numérica**
- A solução ilustra como paralelizar um algoritmo "dividir para consquistar" sujeito apenas ao **requisito de que os subproblemas sejam independentes**

Trabalhadores replicados e bolsa de tarefas



- Dada uma função contínua e positiva **f(x)** e dois valores limites **l** e **r**, onde $l < r$, o problema consiste em computar a área limitada pela função f(x) entre o eixo **x** e as linhas verticais computadas por **l** e **r**: aproximação da integral de f(x) de l a r

Trabalhadores replicados e bolsa de tarefas

Problema da quadratura

- A maneira comum de aproximar a área abaixo da curva é dividir o intervalo [l,r] em uma série de subintervalos, e então usar o cálculo da área de um trapézio para aproximar o valor da área no subintervalo
- Dado o subintervalo [a,b], uma aproximação da área abaixo de **f** de **a** a **b** é a área do trapézio com base **(b-a)** e lados de altura **f(a)** e **f(b)**

Trabalhadores replicados e bolsa de tarefas

Problema da quadratura: abordagem dinâmica

- O problema começa com o intervalo [l,r] e computa-se o ponto do meio **m**
- Calcula-se a área dos três trapézios, compara-se a área do maior trapézio com a soma das áreas dos trapézios menores, se elas forem suficientemente próximas, considera-se a área do maior trapézio como uma aproximação aceitável da área de **f**
- Caso contrário, o processo é repetido para resolver os dois subproblemas de computar as áreas de [l,m] e [m,r]
- Esse **processo é repetido recursivamente** até a solução de cada subproblema convergir

Trabalhadores replicados e bolsa de tarefas

Paralelização de problema "dividir para conquistar"

- Idéia básica: ter um **processo admin** e vários **processos trabalhadores**
- O admin gera o primeiro problema e aguarda os resultados
- Os trabalhadores resolvem os subproblemas e geram novos subproblemas (quando necessário), compartilhando um único canal ("bolsa de tarefas")
- O número de trabalhadores pode ser variável

Trabalhadores replicados e bolsa de tarefas

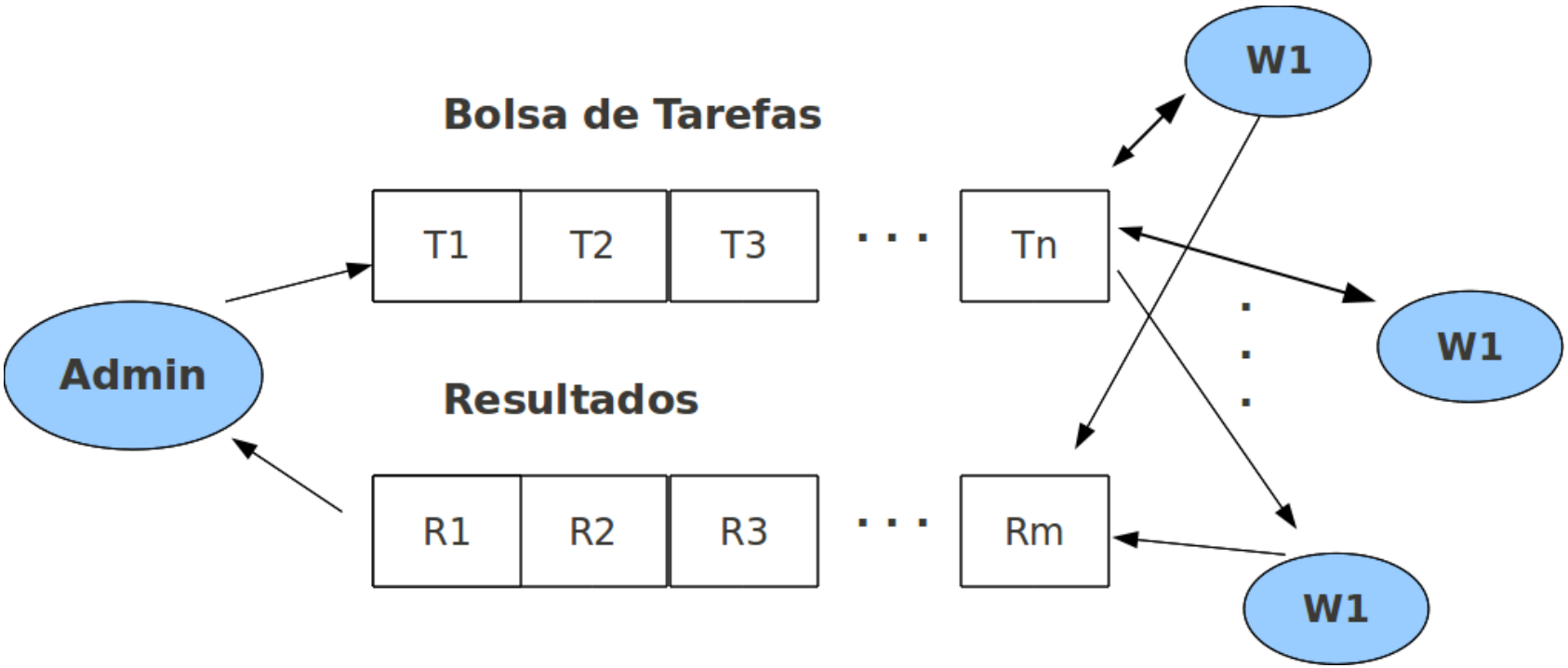
```
chan bag(a,b,fa,fb,area: real)
chan result(a,b,area: real)

[Admin::]
var l,r,fl,fr,a,b,area,total: real
fl := f(l); fr := f(r);
area := (fl+fr)*(r-l)/2;
send bag(l,r,fl,fr,area);
while(area não computada ainda) {
   receive result(a,b,area);
   total := total + area;
   //armazena que calculou a área de [a,b]
}
```

Trabalhadores replicados e bolsa de tarefas

```
[Worker[1:n]::]
var a,b,m,fa,fb,fm: real
var larea, rarea, tarea, diff: real
while(true) {
   receive bag(a,b,fa,fb,tarea);
   m := (a+b)/2; fm := f(m);
   //computa larea e rarea usando trapézios
   diff := tarea - (larea + rarea);
   if (diff é baixo) {
      send result(a,b,tarea);
   else {
      send bag(a,m,fa,fm,larea);
      send bag(m,b,fm,fb,rarea);
   }
}
```

Trabalhadores replicados e bolsa de tarefas



Referências bibliográficas

1. *Concurrent Programming — Principles and Practice*, Andrews, Addison-Wesley, 1991
2. *Programming Language Pragmatics*, Scott, Morgan-Kaufmann, ed. 2, 2006
3. *Operating Systems – Internals and Design Principles*, Stallings, Pearson, ed. 6, 2009
4. *Modern Multithreading*, Carver e Tai, Wiley, 2006
5. *Foundations of Multithreaded, Parallel, and Distributed Programming*, Andrews, Addison-Wesley, 2000
6. *Designing and Building Parallel Programs: concepts and tools for parallel software engineering*, Addison-Wesley, 1995