Programação Concorrente por Troca de Mensagens: I- Continuação

Dimensões da comunicação por mensagens

Unix Windows NT Netware MacOS DOS/VS Vax/VMS Linux Solaris HP/UX AIX Mach Chorus

1. Invocação: explícita vs. implícita
2. Nomeação: directa vs. indirecta
3. Padrão de interacção: 1:1, 1:N, N:1, (N:M)
4. Sincronia vs. Assincronia (operações bloqueantes ou não)
5. Suporte ao não-determinismo (selecção)
6. Fluxo de bytes (stream) vs. fluxo de mensagens
7. Papel dos participantes: simétrico vs. assimétrico (cliente/servidor)

Padrão de interacção 1:1

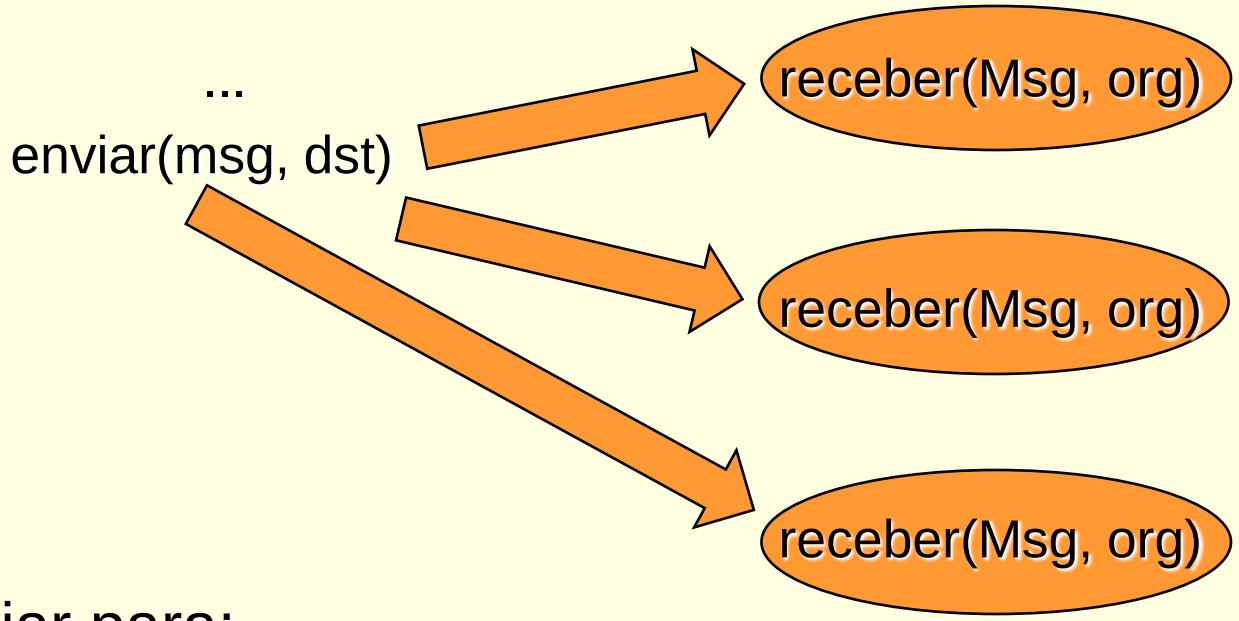
- Fluxo unidireccional:

```
...                                  ...
enviar(msg, dst)   ⟹   receber(Msg, org)
...                                  ...
```

- Fluxo bidireccional:

```
...                                  ...
enviar(msg, dst)   ⟺   receber(Msg, org)
receber(msg, dst)      enviar(Msg, org)
...                                  ...
```

Padrão de interacção 1:N (1)



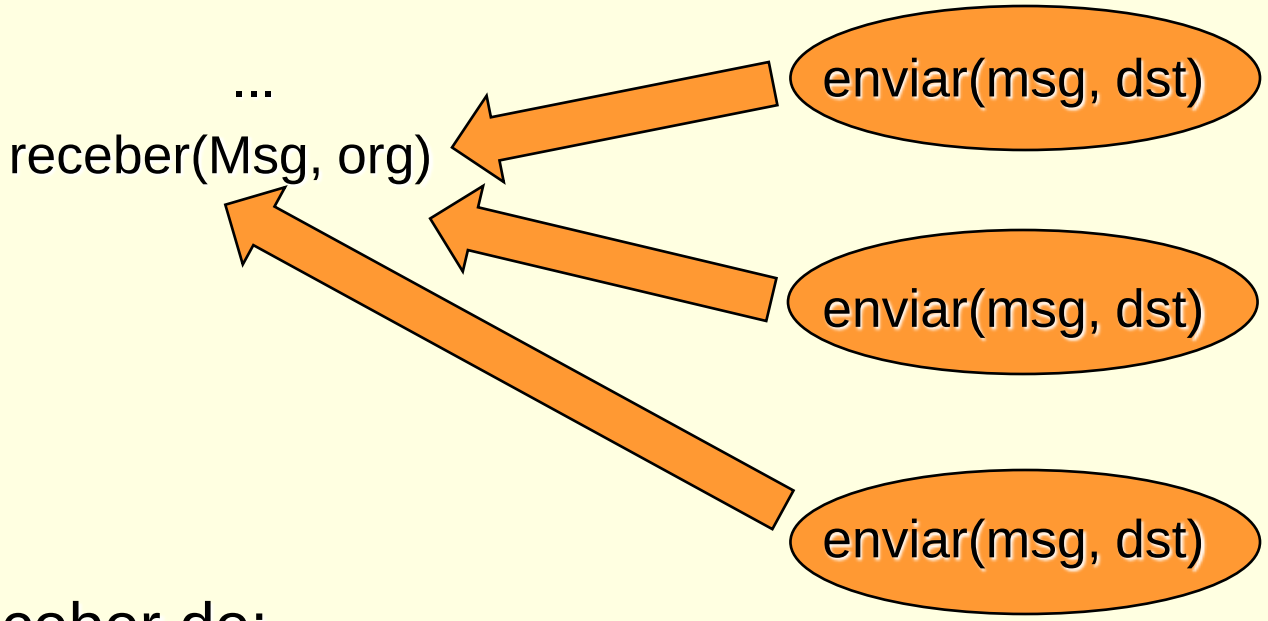
- Enviar para:
  - Todos (broadcast): dst == todos
  - Todos de um grupo de processos (multicast): dst == grupo

.

Padrão de interacção 1:N (2)

- Nomeação implícita ou indirecta
  - O modelo dispõe de primitivas e ou símbolos que permitem indentificar os “alvos”: todos ou grupo.
- Grupos estáticos ou dinâmicos
  - Estáticos: constituição/número de participantes constante e pré-definida no programa.
  - Dinâmicos: processos podem entrar/sair dos grupos, variando a sua constituição
- Exemplos
  - Nenum dos mecanismos de troca de mensagens que abordamos (pipes, message queues) suporta interacções 1:N

Padrão de interacção N:1 (1)



- Receber de:
  - Todos (broadcast): org == todos
  - Todos de um grupo de processos (multicast): org == grupo

.

Padrão de interacção N:1 (2)

- Nomeação implícita ou indirecta, tal como em 1:N.
- Grupos estáticos ou dinâmicos, tal como em 1:N.
- Semântica
  - O receptor aguarda bloqueado até receber as N mensagens, uma de cada emissor. Pode geralmente definir-se uma função que é aplicada ao conjunto de mensagens recebidas, e.g., sum(), avg(), max(), min(), count(), etc.
- Exemplos
  - Nenum dos mecanismos de troca de mensagens que abordamos (pipes, message queues) suporta interacções N:1

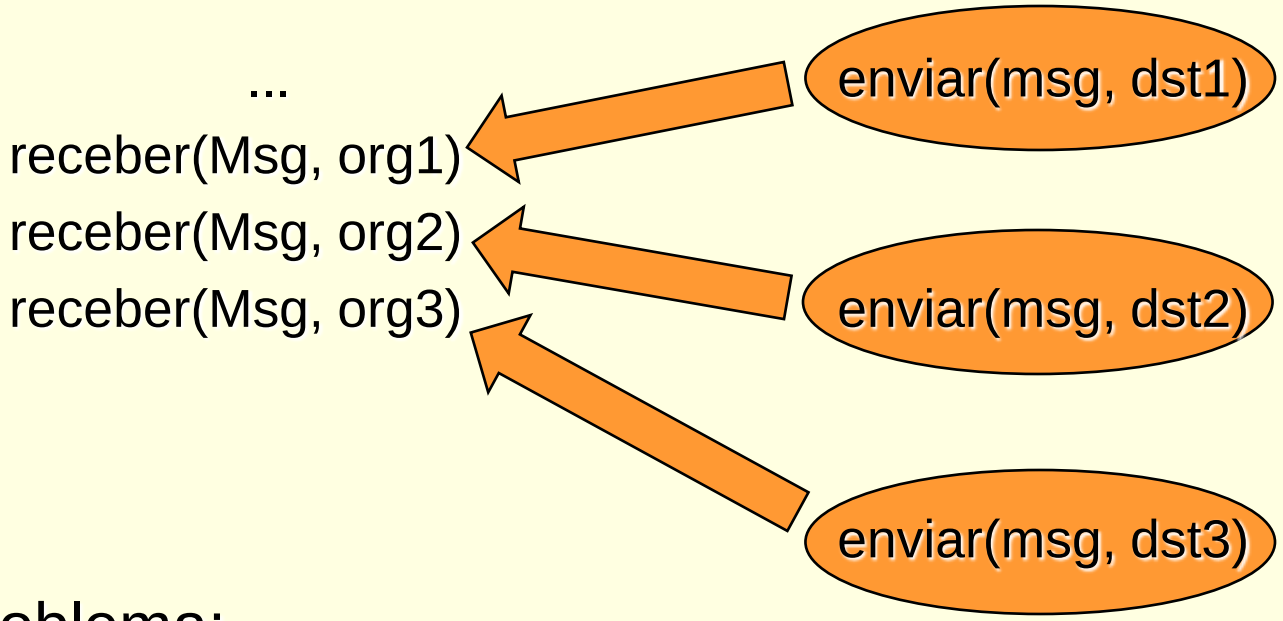
Comunicação síncrona

- Sincronismo nas interacções entre os processos:
  - cada participante espera que o(s) outro(s) termine a operação antes de todos avançarem.
- Suportado em operações bloqueantes:
  - A recepção bloqueia até estar completa; o envio bloqueia até a recepção ter terminado
- Exemplos
  - Os mecanismos de troca de mensagens que abordamos (pipes, message queues) suportam recepção síncrona, mas não envio síncrono. **Para pensar:** como realizar, nesses caos, o sincronismo no envio?

Comunicação assíncrona

- Assincronismo nas interacções entre os processos:
  - Uma operação (de envio ou recepção) retorna imediatamente, sem que isso indique se a interacção com “o outro” já ocorreu.
- Suportado em operações não-bloqueantes:
  - O programador tem ao seu dispôr informação (retorno da função, operação de consulta do estado) que lhe permite saber, quando desejar, se a última operação foi concluída com sucesso ou não.
- Exemplos
  - Os mecanismos de troca de mensagens que abordamos (pipes, message queues) suportam recepção e envio não-bloqueantes.

Suporte ao não-determinismo (1)



- Problema:
  - Como avançar imediatamente com o processamento assim que se receber uma qualquer mensagem?

Suporte ao não-determinismo (2)

- Usando operações não bloqueantes
  - Cada receber() é não-bloqueante; a seguir, verifica-se se recebeu de facto, e, se sim, processa-se; se não, avança-se para o próximo. Desta forma consegue resolver-se o problema, mas está-se a usar espera activa, o que, quando não há clientes activos, representa um desperdício de CPU

```
if ( receber(Msg, org1) == OK ) processa();
else if ( receber(Msg, org2) == OK ) processa();
else if ( receber(Msg, org3) == OK ) processa();
```

Suporte ao não-determinismo (3)

- Usando uma função de selecção
  - Uma função de selecção é uma operação bloqueante que permite monitorar um conjunto de filas, “retornando” quando a uma delas chega uma mensagem. O valor retornado identifica a fila que contém mensagens

```
fila= selecciona(org1, org2, org3);
receber(Msg, fila);
processa();
```

Natureza dos fluxos: estruturados

- Fluxos estruturados:
  - Do ponto de vista aplicacional, existem N tipos de mensagens (N tipicamente reduzido) cada uma representada por uma estrutura de dados; nas operações de envio/recepção são passadas essas estruturas, e o sistema de transporte subjacente transporta essas mensagens como unidades indivisíveis.

char msg [1024];

enviar(msg, dst)  receber(msg, org)

Natureza dos fluxos: não-estruturados

- Fluxos não-estruturados (byte stream):
  - Ao contrário do caso anterior, o sistema de transporte de mensagens não as trata como unidades, mas como sequências de bytes. Isto significa que o programador pode enviar uma mensagem de dimensão X e, ao receber, pedir para receber X bytes mas em vez disso receber uma quantidade Y < X, .

```
                    char msg [1024];
                                        left= 1024;
enviar(msg, 1024, dst)  ------------>   while (left) {
                                         r=receber(msg, left, org);
                                          left= left – r;
                                        }
```

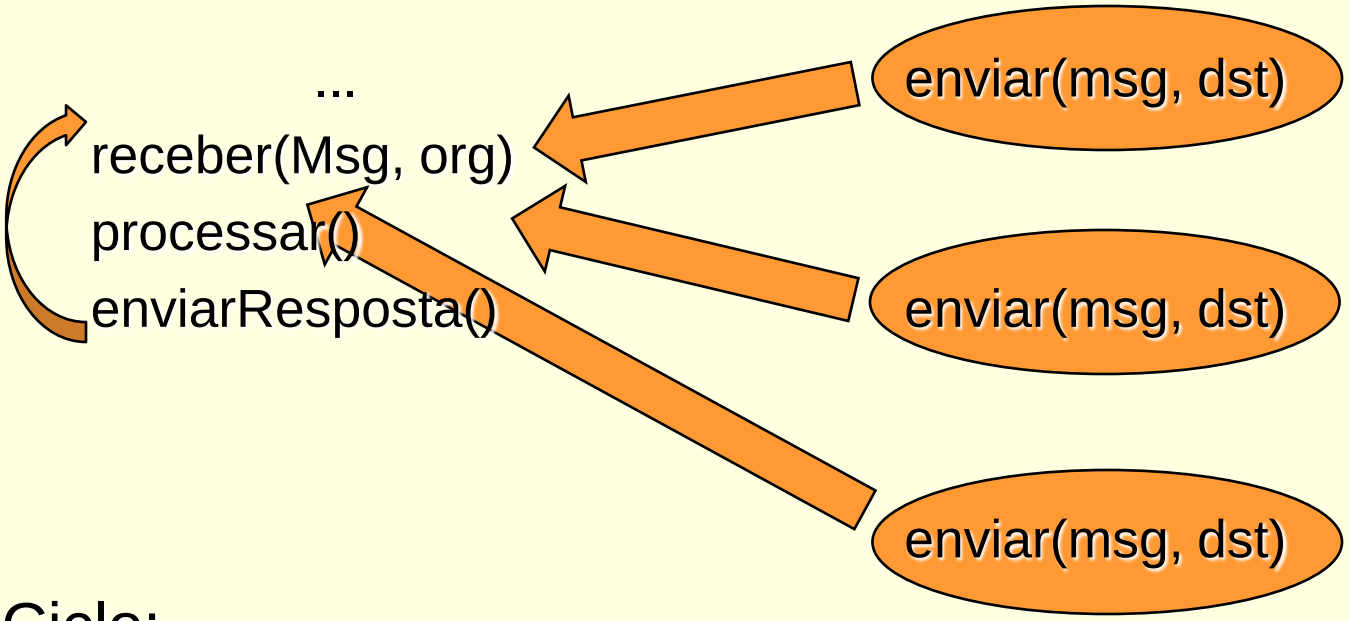
Natureza dos fluxos

- Exemplos
  - Os mecanismos de troca de mensagens que abordamos (pipes, message queues) baseiam-se em trocas não-estruturadas – o programador deve, na recepção (leitura, no caso dos pipes) verificar que a quantidade de dados recebida corresponde à esperada.

Papel dos participantes

- Participantes simétricos
  - Neste modelo de aplicação todos os processos desempenham as mesmas tarefas – nenhum assume um papel especial quando comparado com os outros.
- Participantes assimétricos
  - Um dos casos mais comuns de assimetria é o das aplicações cliente/servidor, em que um conjunto de processos, os clientes, colocam pedidos a um outro, o servidor, que oss processa, respondendo aos clientes. Outro exemplo é o das aplicações mestre/escravo (master/slave ou master/worker), em que um processo (mestre) distribui trabalho aos outros (workers), que, quando o completam, enviam os resultados ao mestre, recebendo mais trabalho, e continuando o ciclo…

Aplicação cliente/servidor



- Ciclo:
  - Receber(); processar(); EnviarResposta()
  - Fazer o mesmo para o próximo cliente

.