Comunicação em Grupo

Prof. Marcelo Veiga Neves
marcelo.neves@pucrs.br

Introdução

- Comunicação entre processos
  - Memória compartilhada vs memória distribuída
  - Troca de mensagens
- Comunicação *one-to-one*
  - Forma mais simples de comunicação entre processos
  - *point-to-point*, ou *unicast*
- Algumas aplicações necessitam de comunicação entre grupos de processos
  - oferecer facilidades para o programador, atomocidade, tolerancia a falhas, etc.
  - oferecer bom nível de desempenho
    - ex.:multicast em relação a vários fluxos de comunicação unicast

Comunicação em grupo

- Objetivo
  - Envio de mensagem para um grupo de processos através de uma *única operação*
- Grupo de processos
  - "Um grupo é um conjunto de processos cooperantes (que agem juntos) especificados pelo sistema ou por um usuário" [TANENBAUM]
- Propriedades:
  - Mensagem enviada ao grupo é recebida por cada um dos seus membros
  - Grupos devem ser dinâmicos

Comunicação em grupo

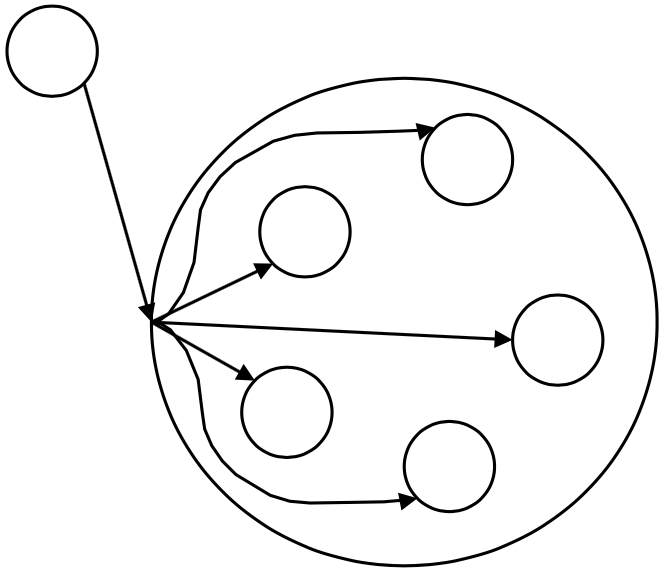
- Formas de comunicação em grupo
  - *One-to-many*
  - *Many-to-one*
  - *Many-to-many*

Comunicação *one-to-many*

- Também chamado *multicast*
  - *Broadcast*: caso especial de *multicast* para todos processos em uma rede (*one-to-all*)
- exemplo:
  - gerente de servidores, todos oferecendo mesmo tipo de serviço
  - o gerente pode mandar mensagem a todos servidores perguntando por um servidor livre para assumir um pedido
  - seleciona primeiro que responde - resposta em unicast
  - gerente de servidores não tem que manter controle sobre servidores livres
- outro exemplo:
  - achar servidor oferecendo um determinado tipo de serviço
  - mensagem em broadcast com “pergunta” - resposta em unicast

Comunicação *one-to-many*

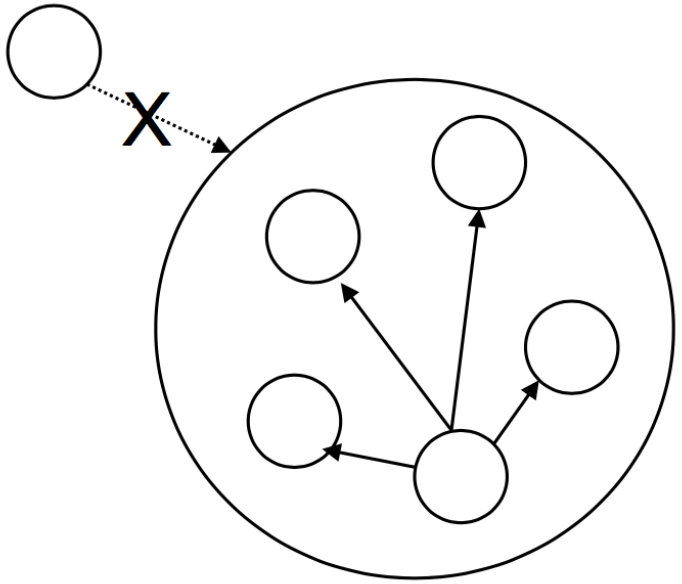
- Gerência de grupos: grupo aberto
  - Qualquer processo pode mandar mensagens para o grupo como um todo
  - exemplo:
    - servidores replicados para processamento distribuído formam grupo aberto pois clientes mandam pedidos para o grupo de servidores



Comunicação *one-to-many*

- Gerência de grupos: grupo fechado
  - Somente membros do grupo podem mandar mensagem para o grupo
  - ex.: grupo de servidores trabalhando em problema comum

Ex.: nodos trocam informações sobre carga, para balanceamento - grupo pode ser fechado pois os nodos trocam informações somente entre eles



Comunicação *one-to-many*

- Gerência de grupos: grupo fechado
  - Problemas de servidor de grupos centralizado:
    - Baixa confiabilidade
    - Baixa “escalabilidade”(potencial para crescer)
  - Uso de replicação do servidor de grupo para resolver tais problemas
    - Overhead aumenta – manter informação dos grupos consistente em todos servidores replicados

Comunicação *one-to-many*

- Sincronização
  - multicast é normalmente assíncrono:
    - não é realístico o transmissor esperar que todos receptores do grupo multicast estejam prontos para receber
    - o transmissor pode não saber quantos receptores existem no grupo

Comunicação *one-to-many*

- Bufferização: *buffered* e *unbuffered multicast*
  - *unbuffered*: mensagem chega e processo receptor não está pronto – SO no receptor descarta mensagem
  - *buffered*: mensagem armazenada para o processo receptor

Comunicação *one-to-many*

- Semântica de envio
  - *send-to-all*: cópia da mensagem é mandada para cada processo do grupo, e a mensagem é armazenada até sua recepção
  - *bulletin-board*: mensagem é endereçada a um canal; processos recebedores copiam mensagem do canal. Condiderada mais flexível pois:
    - a relevância de uma mensagem a um recebedor particular depende do estado do recebedor
    - mensagens não aceitas após um período de tempo podem não ser mais úteis; seu valor depende do estado do enviador

Comunicação *one-to-many*

- Confiabilidade: confirmação de resposta
  - aplicações diferentes tem diferentes requisitos de confiabilidade
  - **0-*reliable***: enviador não espera resposta de nenhum recebedor.
    - **Ex.:** *time signal generator*
  - **1-*reliable***: enviador espera resposta de 1 recebedor - qualquer um.
    - **Ex.:** server manager a procura de um servidor
  - ***m-out-of-n-reliable***: o grupo consiste de n recebedores e o enviador espera uma confirmação de m (1<m<n) dos n recebedores.
    - **Ex.:** algoritmos de consenso por maioria - usados para controle de consistência de informações replicadas usam este tipo de confiabilidade, com m=n/2
  - ***all-reliable***: o enviador espera resposta de todos os recebedores do grupo.
    - **Ex.:** mensagem para atualizar réplicas de arquivo em todos servidores de arquivos envolvidos (grupo)

Comunicação *one-to-many*

- Multicast Atômico
  - confiabilidade *all-reliable*
  - característica *all-or-nothing* - ou entrega mensagem a todos, ou a nenhum
  - método 1:
    - a) transmite a todos;
    - b) espera ack de todos;
    - c) depois de timeout, retransmite aos ainda não confirmados
    - d) volta para b) considerando só os que não confirmaram ainda
    - e) quando todos confirmaram, envio em multicast acabou
  - E se falhas acontecem no transmissor durante o multicast?

Comunicação *one-to-many*

- *Multicast* Atômico
  - Método 2:
    - Enviador
      - Cada mensagem tem um identificador para distingui-la das demais
      - o enviador a manda para o multicast group
      - uso de timeout e retransmissões - em falta de confirmação
    - recebedor:
      - verifica identificador da mensagem para ver se é nova
      - se não for nova, descarta
      - se for nova manda a mesma mensagem para o multicast group em multicast atômico - uso de timeout e retransmissões

Comunicação *one-to-many*

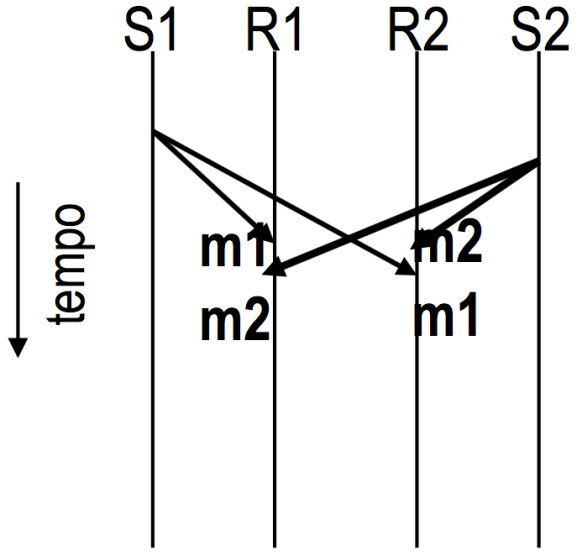
- *Multicast* Atômico
  - Método 2:
    - acontece um *flooding* da mensagem
    - caro – muitas mensagens
      - deve-se optar por multicast atômico somente quando realmente necessário
    - garante que todos processos sobreviventes do grupo "eventualmente" receberão a mensagem, mesmo que o processo enviador falhe durante o multicast
      - "eventualmente" - vai receber, porém não se sabe com que atraso

Comunicação *many-to-one*

- Recepção seletiva e não seletiva
- Não seletiva
  - recebedor quer receber de qualquer um do grupo
- Seletiva
  - recebedor quer controlar dinamicamente de quem receber
  - Exemplo.:
    - processo buffer recebe mensagens de produtores e consumidores;
    - se buffer cheio -> aceitar mensagens só de consumidores
    - se buffer vazio -> aceirar mensagens só de produtores
    - outro caso -> aceitar mensagens dos dois

Comunicação *many-to-many*

- Aspectos já discutidos para one-to-many e many-to-one se incluem + **entrega ordenada de mensagens**
  - entrega das mensagens em ordem aceitável para a aplicação
  - ordenação de mensagens requer mecanismo de sequenciamento (serialização)



Comunicação *many-to-many*

- Problema da ordenação das mensagens
- em *one-to-many*: sequenciar multicasts
  - enviador inicia próximo multicast só depois de acabar o que já está em curso
- em *many-to-one*: mensagens são recebidas pelo recebedor na ordem em que chegam da rede ...
- em *many-to-many?*
  - Vários enviadores e vários recebedores em diversos pontos rede apresenta atrasos diferentes dependendo das posições dos processos ...
  - falhas de links, roteadores ...
  - como garantir mesma percepção de ordem para os vários recebedores?

Comunicação em grupo em Java

- Comunicação *one-to-one*
  - Não confiável: UDP
  - Confiável: TCP
- Comunicação one-to-many
  - Não confiável: IP Multicast
  - Confiável: JGroups

IP Multicast

- Comunicão on-to-many não confiável
- Usa endereços IP da Classe D (*multicast*): 224.0.0.0 até 239.255.255.255
- Classes usadas:
  - MulticastSocket(porta)
    - Métodos: joinGroup(end), receive(pacote), leaveGroup(end), close()
  - DatagramSocket
  - DatagramPacket
  - InetAddress

Exemplo: multcast receive

```
import java.io.*;
import java.net.*;
import java.util.*;

public class recebeAlo {
    public static void main(String[] args) throws IOException {
        MulticastSocket socket = new MulticastSocket(5000);
        InetAddress grupo = InetAddress.getByName("230.0.0.1");
        socket.joinGroup(grupo);
        byte[] entrada = new byte[256];
        DatagramPacket pacote = new DatagramPacket(entrada,entrada.length);
        socket.receive(pacote);
        String recebido = new String(pacote.getData(),0,pacote.getLength());
        System.out.println("Received: "+recebido);
        socket.leaveGroup(grupo);
        socket.close();
    }
}
```

Exemplo: multcast send

```
import java.io.*;
import java.net.*;
import java.util.*;

public class enviaAlo {
    public static void main(String[] args) throws IOException {
        byte[] saida = new byte[256];
        String mens = "Alo, mundo!";
        saida = mens.getBytes();
        DatagramSocket socket = new DatagramSocket();
        InetAddress grupo = InetAddress.getByName("230.0.0.1");
        DatagramPacket pacote = new DatagramPacket(saida,saida.length,grupo,5000);
        socket.send(pacote);
        socket.close();
    }
}
```

JGroups

- *Framework* para comunicação em grupo confiável
- Processos interagem com grupos através de um objeto da classe JChannel
- JChannel age como um manipulador para determinado grupo
  - Um JChannel é criado desconectado
  - A operação connect() vincula o JChannel a determinado grupo (se o grupo não existe, ele é criado)
  - A operação disconnect() abandona o grupo
  - close() encerra o grupo

Exemplo: JGroup receive

```
import org.jgroups.JChannel;
import org.jgroups.ReceiverAdapter;
import org.jgroups.Message;

public class HelloWorldReceiveJG extends ReceiverAdapter {
    public static void main(String[] args) {
        try {
            JChannel channel = new JChannel();
            channel.setReceiver(new ReceiverAdapter() {
                public void receive(Message msg) {
                    System.out.println("received msg from " + msg.getSrc() + ": " + msg.getObject());
                }
            });
            channel.connect("HelloWorld");
            Thread.sleep(10000);
            channel.close();
            System.out.println("done!");
        }
        catch(Exception e) {
        }
    }
}
```

Exemplo: JGroup send

```
import org.jgroups.*;

public class HelloWorldSendJG {
    public static void main(String[] args) {
        try {
            JChannel channel = new JChannel();
            channel.connect("HelloWorld");
            Message msg = new Message (null, null, "Hello!");
            channel.send(msg);
            channel.disconnect();
        }
        catch (Exception e) {
        }
    }
}
```

Referências

- Material baseado em slides dos Profs. Roland Teodorowitsch, Avelino Zorzo, Celso Costa, Fernando Dotti e Luiz Gustavo Fernandes
- E nos seguintes livros:
  - Distributed Systems: Principles and Paradigms, Andrew S. Tanenbau