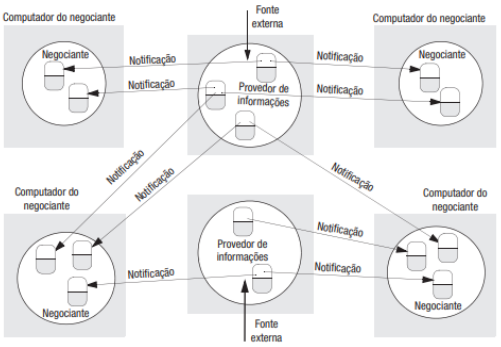
Sistemas Publicar-Assinar

Dividida em dois componentes: **publicadores e assinantes**.

- **Publicadores** divulgam um evento.
  Responsáveis pela publicação dos eventos.
- **Assinantes** expressam interesse em eventos por meio de assinaturas.
  Assinaturas são sequências de caracteres ou de texto que determinam exatamente a que tipo de evento que se tem interesse.
- O sistema entrega notificações relacionadas ao evento a seus assinantes.

Aplicações de sistemas publicar-assinar

- Sistema de informação financeira
  Um comerciante que venda arroz, por exemplo, pode publicar um evento chamado arroz e os negociantes assinantes são avisados desta publicação.
    - Sistemas de sala de negociações



Aplicações de sistemas publicar-assinar

EXEMPLOS

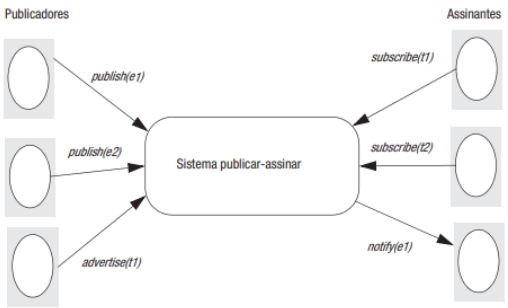
- Divulgação de dados em tempo real (feed RSS)
  As notícias da internet usam esse conceito.
  feed RSS: envia notícias com base no interesse relacionado da pessoa usando o sistema publicar-assinar.
- Trabalho cooperativo
  Quando vários participantes precisam ser notificados sobre eventos ocorrendo.
- Computação ubíqua (eventos de localização)
  onde pequenos dispositivos do ambiente precisam se comunicar. Por exemplo, na chegada de uma pessoa ao ambiente os dispositivos são notificados para realizarem suas tarefas.
- Aplicativos de monitoramento
  Permite a configuração de um ambiente, para que todos

elementos sejam avisados sobre alguma mudança.

Características do sistemas publicar-assinar

- Heterogeneidade
  Comunicação heterogênea, não precisam ser implementados no mesmo ambiente ou juntos.
- Assíncronos
  Os publicadores não precisam estar sincronizados com assinantes.
- Garantias de entrega
  Dependendo da necessidade, pode-se garantir. Exemplo: no comércio é interessante que todos recebam.
- Garantias de tempo real (monitoramento hospitalar)
  O aviso deve acontecer em tempo curto.

Modelo de Programação



De um lado se tem os **publicadores** que usam a função **publish** para enviar publicações e **notify** para notificar e os **assinantes** que usam **subscribe** para serem avisados sobre eventos de interesse deles.

⚠ Explicar melhor a função **advertise** na classe de publicadores**.**

Modelos de assinatura

Pode ser baseada em diferentes padrões.

- Baseado em canal
  Os publicadores publicam em um canal específico, semelhante a comunicação indireta dos JGroups.
- Baseado em tópico/assunto
  Mensagens filtradas por tópico.
- Baseado em conteúdo
  Há um refinamento maior das mensagens, publicadores publicam eventos específicos. Pode-se fazer associação de várias assinaturas:
  - exemplo: comunicação indireta E com autor específico.
- Baseado em tipo: estratégia baseada em objetos.
  Assinaturas definidas por um tipo específico de objeto (atributos ou métodos).

- Baseado em objetos de interesse.
  O filtro da assinatura pode ser definido com base nas mudanças do objeto.
  - exemplo: evento de clique de um botão.
- Baseado em contexto.
  Pode-se levar em consideração circunstâncias físicas.
  - exemplo: um sensor gera um evento para objetos com interesse.

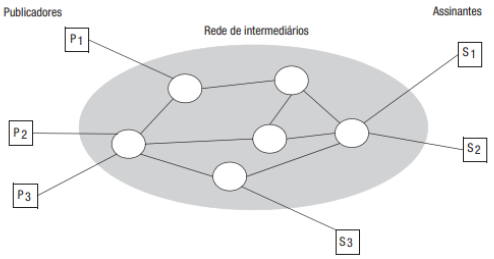
Problema de implementação

- Objetivo: garantir que os eventos sejam entregues eficientemente para todos os assinantes que tenham definido filtros que correspondam ao evento.
  Quando um publicador envia um evento, todos assinantes devem receber o evento.
- Implementações centralizadas versus distribuídas
  Estratégias para garantir a entrega:
  - **Centralizada:** Simples de implementar, mas não apresenta flexibilidade e escalabilidade.
    Uma entidade central é responsável por receber todos os eventos, fazer o filtro desses eventos e entregar para os assinantes.
    Desvantagem: não é flexível e não possui escalabilidade.

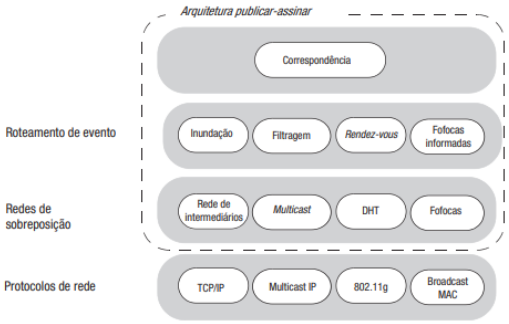
    ⚠ Pedir explicação mais clara sobre a **desvantagem da abordagem centralizada**.

  - **Distribuída:** Implementação totalmente peer-to-peer: não há distinção entre publicadores, assinantes e intermediários.
    Há uma série de nós organizados na estrutura de ponto a ponto, onde existem diversos nós intermediários (**rede de intermediários**) que garantem a chegada das notificações.

Implementação Distribuída



Arquitetura de Sistemas Global



Na implementação da estratégia distribuída há preocupações adicionais.

Correspondência: garantir que o assinante receba exatamente o que pediu.

Rede de sobreposição: Para que o roteamento ocorra, é necessária uma interconexão entre nós (rede de sobreposição). Pode-se usar 4 estratégias:

- **Rede intermediária:** Nós conectados uns aos outros.
- **Multicast:** Garante que as mensagens sejam entregues para todos nós ou um grupo seleto.
- **DHT:** Tabela hash semelhante à tabela hash da rede bittorrent.
- **Fofocas:** Estratégia em que os nós vizinhos trocam entre si eventos que ocorreram até chegar ao nó assinante.

Toda a arquitetura funciona sobre redes já existentes (**Protocolos de rede**).

Arquitetura de sistemas global

- Estratégias de roteamento baseado em conteúdo:
  - Inundação
    É a mais simples, é apenas um broadcast, os nós de uma rede intermediária recebem todas as mensagens direcionadas aquela rede.
    Desvantagem: gera um tráfego muito grande.
  - Filtragem
    Reduz o tráfego, os nós só encaminham publicações nos caminhos onde houver assinantes válidos.
  - Anúncios:
    Melhoramento da filtragem

- Rendez-vous

Filtragem

- Os intermediários só encaminham notificações na rede no caminho em que houver assinantes válidos

```
upon receive publish(event e) from node x                    1
    matchlist := match(e, subscriptions)                     2
    send notify(e) to matchlist;                             3
    fwdlist := match(e, routing);                            4
    send publish(e) to fwdlist - x;                          5
upon receive subscribe(subscription s) from node x           6
    if x is client then                                      7
        add x to subscriptions;                              8
    else add(x, s) to routing;                               9
    send subscribe(s) to neighbours - x;                     10
```

Rendez-Vous

- Repartir a responsabilidade pelo espaço de eventos entre o conjunto de intermediários na rede.

  Cada subconjunto de nós dentro da rede de intermediários é responsável por um subconjunto de eventos, para balancear a carga.

- Nós de rendez-vous são os nós intermediários responsáveis por determinado subconjunto do espaço de eventos.
- Função SN(s): retorna um ou mais nós de rendez-vous responsáveis pela assinatura s.

  Cada nó intermediário tem como descobrir quem são os nós responsáveis por eventos com assinatura “s”.
- Função EN(e): retorna um ou mais nós rendez-vous responsáveis por corresponder o evento **e** às assinaturas.

  retorna os nós responsáveis por fazer a correspondência entre um evento **”e”** e sua assinatura.

```
upon receive publish(event e) from node x at node i
    rvlist := EN(e);
    if i in rvlist then begin
        matchlist <- match(e, subscriptions);
        send notify(e) to matchlist;
    end
    send publish(e) to rvlist - i;
upon receive subscribe(subscription s) from node x at node i
    rvlist := SN(s);
    if i in rvlist then
        add s to subscriptions;
    else
        send subscribe(s) to rvlist - i;
```

Divide melhor as responsabilidades de assinatura e publicação.

Exemplos de Sistema Publicar-Assinar

Sistemas (leituras recomendadas)	*Modelo de assinatura*	*Modelo de distribuição*	*Roteamento de eventos*
CORBA Event Service (Capítulo 8)	Baseado em canal	Centralizado	–
TIB Rendezvouz [Oki *et al.* 1993]	Baseado em tópicos	Distribuído	Filtragem
Scribe [Castro *et al.* 202b]	Baseado em tópicos	*Peer-to-peer* (DHT)	*Rendez-vouz*
TERA [Baldoni *et al.* 2007]	Baseado em tópicos	*Peer-to-peer*	Fofoca informada
Siena [Carzaniga *et al.* 2001]	Baseado em conteúdo	Distribuído	Filtragem
Gryphon [www.research.ibm.com]	Baseado em conteúdo	Distribuído	Filtragem
Hermes [Pietzuch e Bacon 2002]	Baseado em tópico e em conteúdo	Distribuído	*Rendez-vouz* e filtragem
MEDYM [Cao e Singh 2005]	Baseado em conteúdo	Distribuído	Inundação
Meghdoot [Gupta *et al.* 2004]	Baseado em conteúdo	*Peer-to-peer*	*Rendez-vouz*
Structure-less CBR [Baldoni *et al.* 2005]	Baseado em conteúdo	*Peer-to-peer*	Fofoca informada

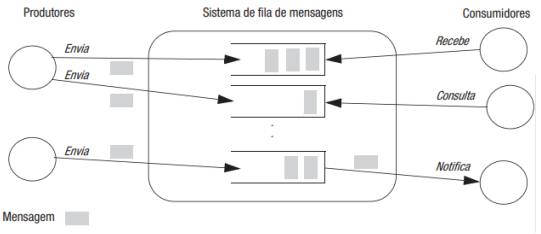
---

Filas de mensagem

A comunicação é do tipo **um para um**. Comunicação em grupo e publicar e assinar são do tipo 1 para n normalmente.

- Serviço de comunicação indireta ponto a ponto;

- Modelo de programação
  - Produtores enviam mensagens para uma fila e os consumidores recebem mensagens dessa fila.
  - Estilos de recepção suportados:
    - Recepção com bloqueio
      Um processo que quer receber determinada mensagem da fila fica bloqueado até receber essa mensagem.
    - Recepção sem bloqueio
      O receptor consulta a fila e verifica se existe alguma mensagem a ser recebida, não tendo continua a funcionar normalmente.
    - Notificação
      O receptor é avisado pela fila de mensagem assim que chegar uma mensagem.



Modelo de programação

- As mensagens são persistentes
  Garante que o consumidor possa receber a mensagem em qualquer momento. São gravadas em meios persistentes (banco de dados, arquivos...).
- Muitos suportam o envio de mensagens dentro de uma transação.
  Garante que seja enviada toda ou de jeito nenhum, sem envios parciais.
- Suportam também a transformação de mensagens.
  A possibilidade de modificar uma mensagem recebida.
- Algumas implementações suportam segurança.
  Usam o protocolo SSL.
- As filas de mensagens possuem um desacoplamento espacial e temporal.
  Produto e consumidor não precisam saber quando um e outro se encontram e não precisam existir ao mesmo tempo (pois as mensagens

são armazenadas em meios persistentes).

O JMS(Java Messaging Service)

Exemplo de implementação.

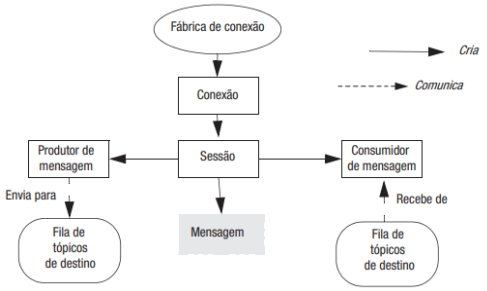
- Especificação Java para comunicação indireta (publicar-assinar e fila de mensagens);

- Funções no JMS

  Tem-se 4 elementos:

  - Cliente: produz ou consome mensagens. produtores e consumidores.
  - Provedor: Sistemas que implementam a especificação JMS
  - Mensagem: objeto usado para comunicação entre clientes JMS
  - Destino: Objeto que suporta comunicação indireta.

Publicação com JMS



Toda aplicação JSM tem a **fábrica de conexões**, que vai criar uma ou mais conexões, as **conexões** podem criar uma ou mais sessões, e as **sessões** podem criar produtores, consumidores e mensagens. Os **produtores** enviam e os **consumidores** recebem mensagens da fila de tópicos.

Exemplo FireAlarmJMS

```
import javax.jms.*;
import javax.naming.*;
public class FireAlarmJMS {
    public void raise( ) {
    try {                                                          1
        Context ctx = new InitialContext( );                       2
        TopicConnectionFactory topicFactory =                      3
            (TopicConnectionFactory)ctx.loo kup("TopicConnectionFactory");  4
        Topic topic = (Topic)ctx.lookup("Alarms");                 5
        TopicConnection topicConn =                                6
            topicFactory.createTopicConnection( );                 7
        TopicSession topicSess = topicConn.createTopicSession(false,  8
                                    Session.AUTO_ACKNOWLEDGE);     9
        TopicPublisher topicPub = topicSess.createPublisher(topic);    10
        TextMessage msg = topicSess.createTextMessage( );          11
        msg.setText("Fire!");                                      12
        topicPub.publish(msg);                                     13
        } catch (Exception e) {                                    14
    }                                                              15
}
```

Exemplo FireAlarmConsumerJMS

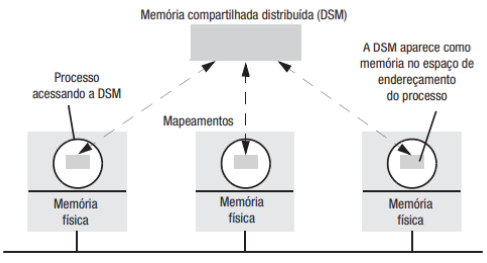
```
import javax.jms.*;
import javax.naming.*;
public class FireAlarmConsumerJMS {
    public String await( ) {
    try {                                                          1
        Context ctx = new InitialContext( );                       2
        TopicConnectionFactory topicFactory =                      3
            (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");  4
        Topic topic = (Topic)ctx.lookup("Alarms");                 5
        TopicConnection topicConn =                                6
            topicFactory.createTopicConnection( );                 7
        TopicSession topicSess = topicConn.createTopicSession(false,   8
                                    Session.AUTO_ACKNOWLEDGE);     9
        TopicSubscriber topicSub = topicSess.createSubscriber(topic);  10
        topicSub.start( );                                         11
        TextMessage msg = (TextMessage) topicSub.receive( );       12
        return msg.getText( );                                     13
        } catch (Exception e) {                                    14
            return null;                                           15
    }                                                              16
}
```

Memória compartilhada distribuída

Estratégia de comunicação indireta

Abstração usada para compartilhar dados entre computadores que não compartilham memória física

A ideia é garantir a comunicação indireta através da reunião das memórias físicas de diversas máquinas em uma única grande memória. Sempre que se escreve na memória compartilhada distribuída, o sistema se encarrega de escrever a mensagem na memória física adequada.

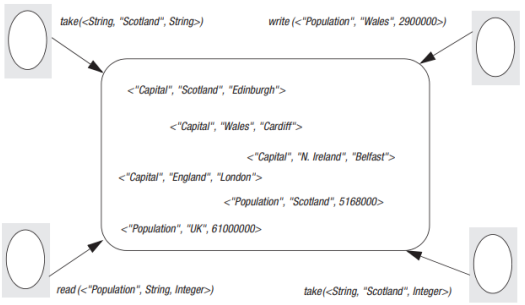


Espaço de tuplas

- Processos se comunicam indiretamente, colocando tuplas em um espaço de tuplas, enquanto outros processos podem ler ou remover tuplas desse espaço

- Tuplas consistem em uma sequência de um ou mais campos de dados tipados, como <"fred", 1958>.
  Tem-se processos que vão escrever e processos que vão ler as tuplas.
- Operações: read, take, write.

Espaço de tuplas



take: retira do espaço de tuplas qualquer tupla que possua determinado padrão.

write: escreve uma tupla no espaço de tuplas.

read: lê tuplas baseando-se em um padrão.