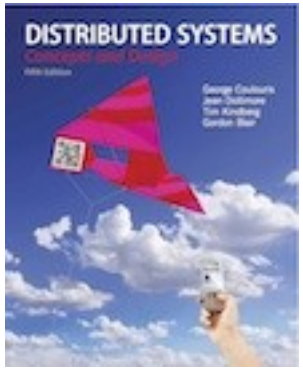
Capítulo 6
Comunicação Indireta



From Coulouris, Dollimore, Kindberg and Blair

Distributed Systems:
Concepts and Design

Edition 5, © Addison-Wesley 2012

Comunicação Indireta

- Definida como comunicação entre duas entidades de um Sistema Distribuído por meio de um intermediário, sem acoplamento direto entre remetente e destinatário
- Frequentemente usado em SDs em que são previstas alterações
- Ambientes móveis – cliente ou servidor se conectam e desconectam frequentemente
- Disseminação de eventos distribuídos
- Há uma maior sobrecarga no desempenho devido ao nível de indireção acrescentado e podem ser mais difíceis de gerenciar

Acoplamento Temporal e Espacial em SDs

	Acoplamento temporal	*Desacoplamento temporal*
Acoplamento espacial	*Propriedades:* comunicação direcionada para determinado destinatário (ou destinatários); o destinatário (ou destinatários) deve existir nesse momento no tempo. *Exemplos:* passagem de mensagens, invocação remota (consulte os Capítulos 4 e 5).	*Propriedades:* comunicação direcionada para determinado destinatário (ou destinatários); o remetente (ou remetentes) e o destinatário (ou destinatários) podem ter tempos de vida independentes. *Exemplos:* consulte o Exercício 6.3.
Desacoplamento espacial	*Propriedades:* o remetente não precisa conhecer a identidade do destinatário (ou destinatários); o destinatário (ou destinatários) deve existir nesse momento no tempo. *Exemplos: multicast* IP (consulte o Capítulo 4).	*Propriedades:* o remetente não precisa conhecer a identidade do destinatário (ou destinatários); o remetente (ou remetentes) e o destinatário (ou destinatários) podem ter tempos de vida independentes. *Exemplos:* a maioria dos paradigmas de comunicação indireta abordados neste capítulo.

Figura 6.1 Acoplamento espacial e temporal em sistemas distribuídos.

Comunicação em Grupo

Enviar uma simples mensagem de um processo para cada processo membro de um grupo de processos.

Antes de continuar - revisão

- Unicast/Multicast/Broadcast
- Mensagem para: Um/Grupo/Todos

Útil para:

- Tolerância a falhas baseado em serviços replicados.
- Descoberta de servidores em redes espontâneas.
- Melhor performance e replicação de dados.
- Propagação de notificação de eventos.

Comunicação em Grupo – IP multicast

- Usa a estrutura do protocolo IP.
- Permite a transmissão de um único pacote IP para um conjunto de computadores
- Um grupo multicast é um endereço IP classe D.
- IP multicasting só está disponível para comunicação UDP.
- Modelo de Falha: Não é confiável.

Comunicação em Grupo/Multicast - Dinamismo

- Criação e destruição de grupos
- entrada e saída de processos de um grupo

- Multiplicidade
- processo pode fazer parte de diversos grupos simultaneamente

- Abstração e transparência
- os processos se comunicam com o grupo, e não com diversos processos

- Implementação depende do HW
- multicast, broadcast

Comunicação em Grupo - Características

- Grupos abertos / grupos fechados
- permitir / proibir que um processo não pertencente ao grupo envie uma mensagem para o grupo
- acesso a servidores replicados / computação paralela para resolver um problema

- Grupos peer / grupos hierárquicos
- decisões coletivas / existe uma hierarquia de processos
- simétrico sem ponto único de falha / ponto único de falhas (coordenador)
- tomada de decisão mais complexa / coordenador decide pelo grupo

Comunicação em Grupo - Características (2)

- Níveis de Confiabilidade
- Atômica: a mensagem é recebida por todos os processos, ou por nenhum processo
- Confiável: é feito o possível para enviar a mensagem para todos os processos, mas isto não é garantido
- Não-confiável: é enviada somente uma mensagem sem garantia de recebimento
- Apenas a última é suportada no IP Multicast

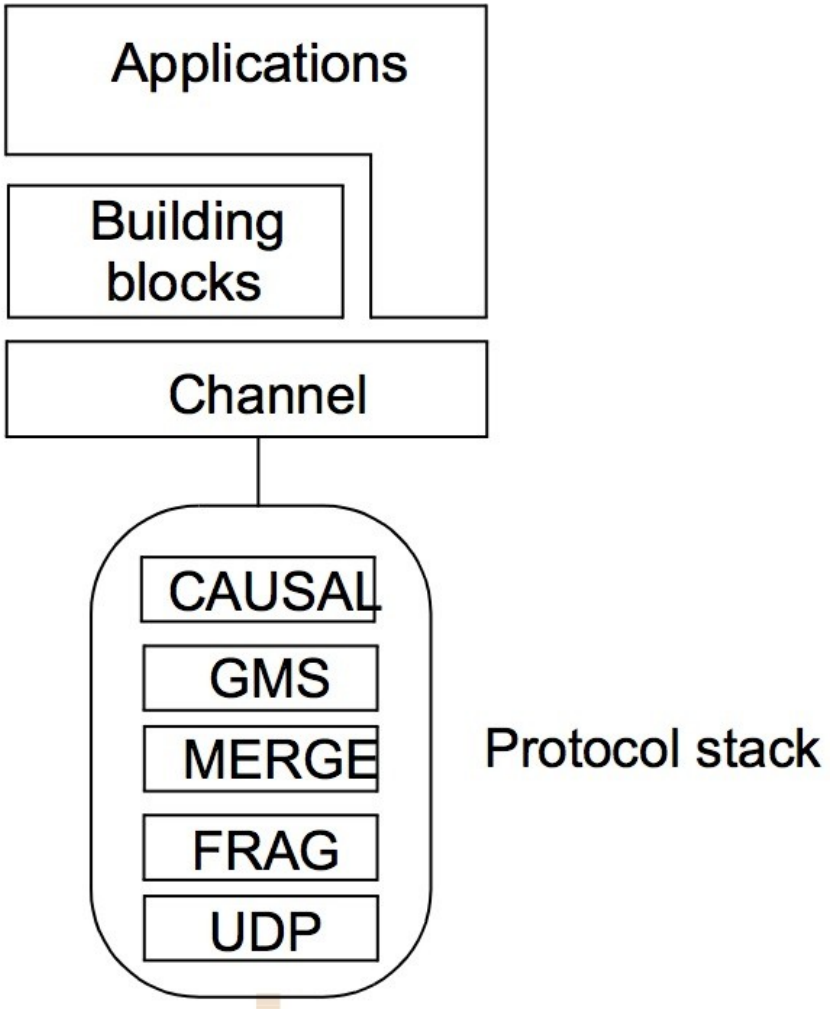
Comunicação em Grupo - Características (3)

- Tipos de Ordenação
- Ordenação absoluta: as mensagens são recebidas na mesma ordem em que foram enviadas.

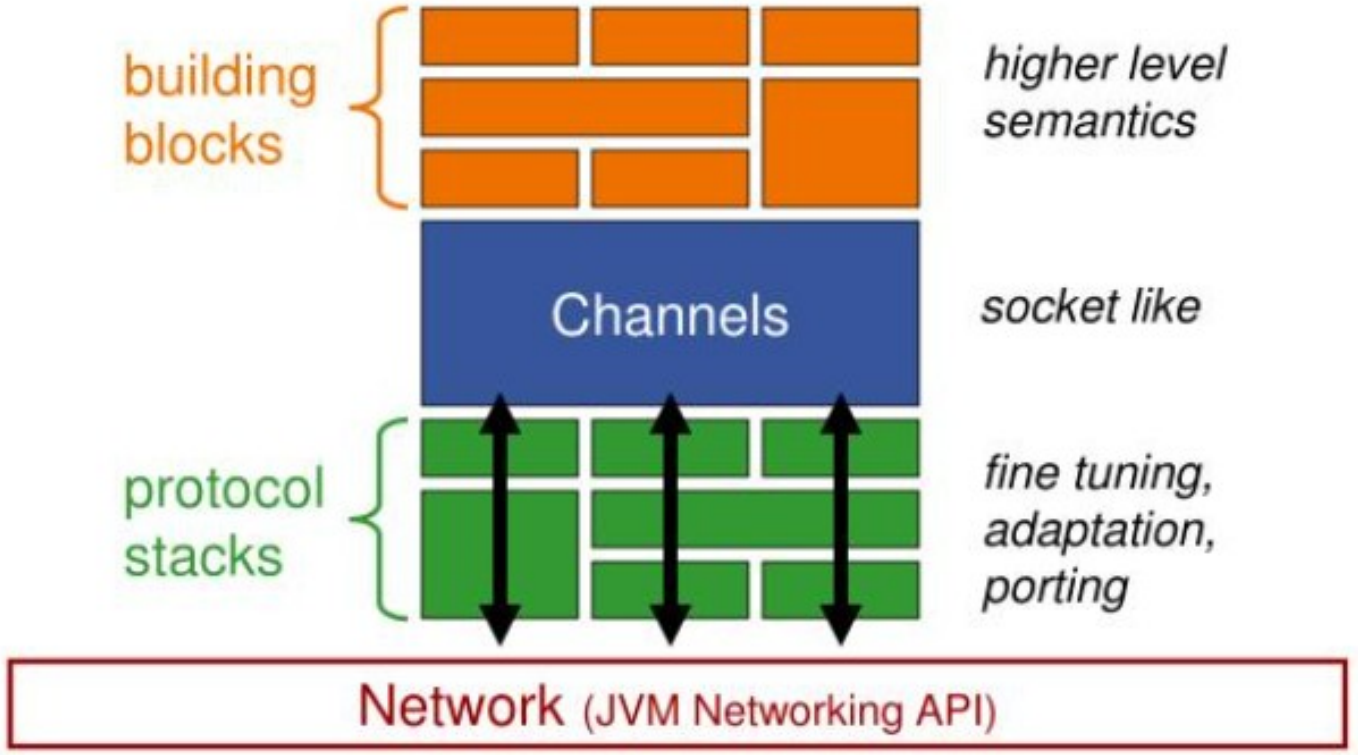
- Ordenação consistente: a ordem de recebimento das mensagens é a mesma para todos os participantes do grupo, mesmo se não for a ordem de envio.

- Ordenação de causa: se o envio de uma mensagem causa o envio de outra mensagem, então as mensagens são enviadas aos participantes do grupo na mesma ordem.

Arquitetura do JGroups

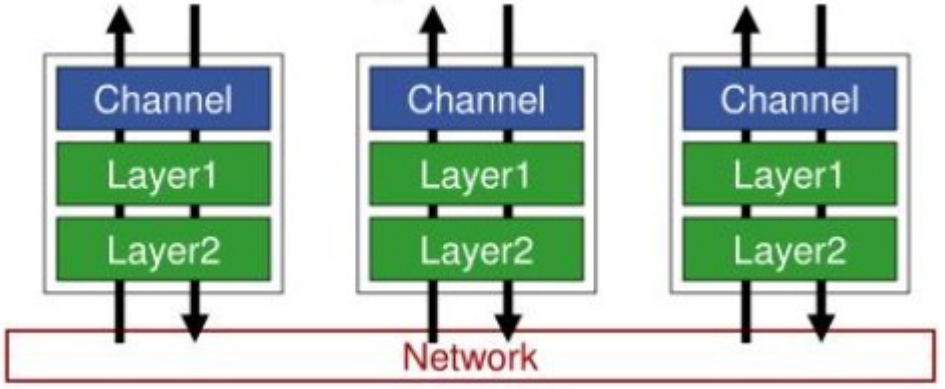


Arquitetura do JGroups



Trabalho - Chat usando JGroups

- Cada instância de um canal se situa sobre uma pilha de protocolos
- A pilha é definida por uma string de propriedades
- `Channel meuCanal = new JChannel("LAYER1:LAYER2");`



Trabalho - Chat usando JGroups

- `Channel myChan = new JChannel("UDP:PING:FD:GMS");`
  - A pilha contém Camadas UDP, PING, FD, and GMS (bottom-up)
  - Corresponde às classes : `org.jgroups.protocols.UDP`
    `org.jgroups.protocols.PING`
    `org.jgroups.protocols.FD`
    `org.jgroups.protocols.GMS`
  - **UDP**: IP multicast transport based on UDP
  - **PING**: initial membership (used by GMS)
  - **FD**: Failure detection (heartbeat protocol)
  - **GMS**: Group membership protocol.

Java class *JGroupsCluster*

```
public class JGroupsCluster
{
   public static void main( String[] args ) throws Exception {
   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("MyCluster");
      int i = 0;
      while (i  < 60) {
         channel.send(new Message(null, "hello world"));
         Thread.sleep(1000);
      }
      channel.close();
   }
}
```

Trabalho - Chat usando JGroups

- Considerando o exemplo do Slide Anterior, faça um programa de chat que permita dois ou mais usuários se comunicarem em um chat online.
- O programa deve solicitar que o usuário digite seu nome ao iniciar a aplicação.
- O programa deve terminar quando o usuário digitar “sair” no console.
- Todas as mensagens digitadas pelos os usuários devem ser do formato:

```
29/03/2017 12:30 Rodrigo: Mensagem
```

Comunicação Persistente

- A mensagem deve ser armazenada enquanto ela aguarda para ser enviada para o destino
- Ex: E-mails ou secretárias eletrônicas
- A secretária eletrônica persiste a mensagem até que ela seja ouvida pelo receptor e retirada da fila das mensagens
- Sem esse mecanismo, é necessário que o receptor esteja sempre disponível
- Isso não é válido para toda e qualquer situação.

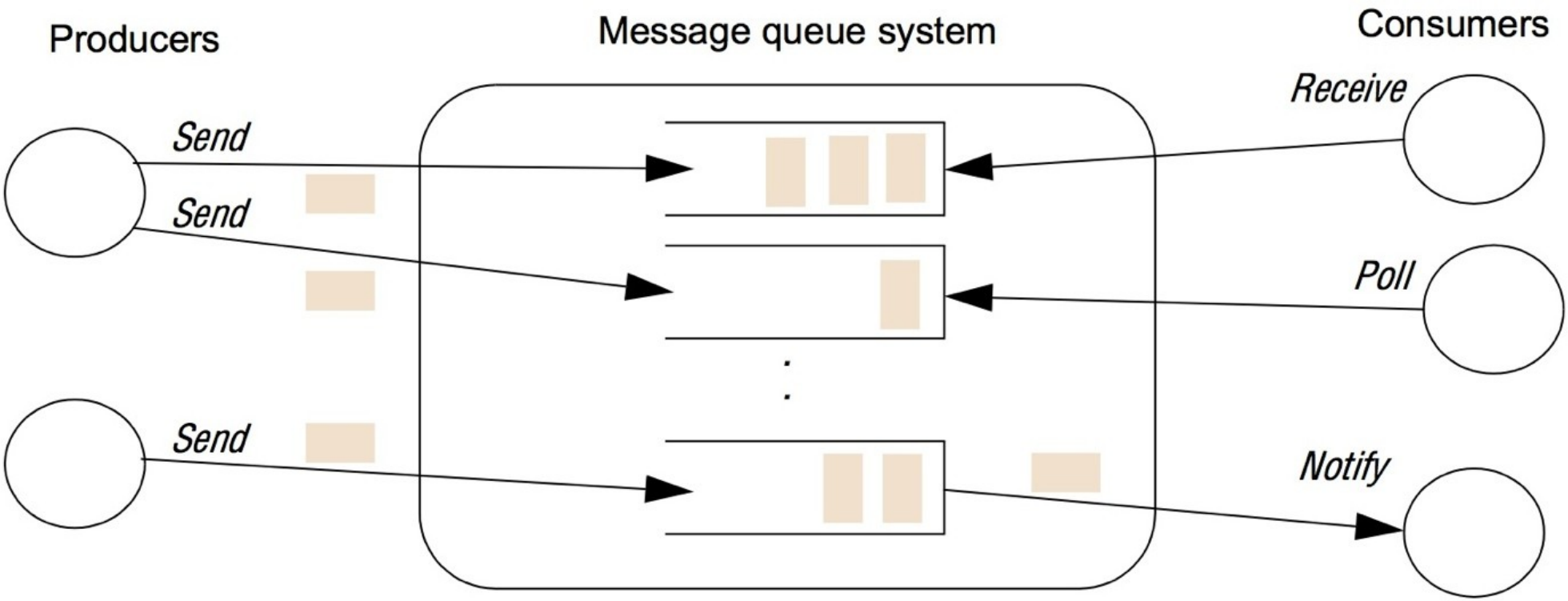
Message Oriented Middleware

- Middleware Orientado a Mensagens ou MOM
- Aplicações se comunicam através do envio de mensagens para filas ou tópicos específicos.
- As filas são processadas e as mensagens são enviadas para outros computadores
- Podem haver vários passos intermediários.
- A mesma mensagem pode ser entregue a vários receptores
- O MOM garante a entrega.
- Processar e avisar o transmissor fica por conta do processo receptor

MOM

- Mensagens podem conter qualquer tipo de infomação
- Deve haver um identificador único para cada fila de recepção
- Filas são gerenciadas por gerenciadores de filas
  - Que também podem funcionar como buffers

Middleware Orientado a Mensagens



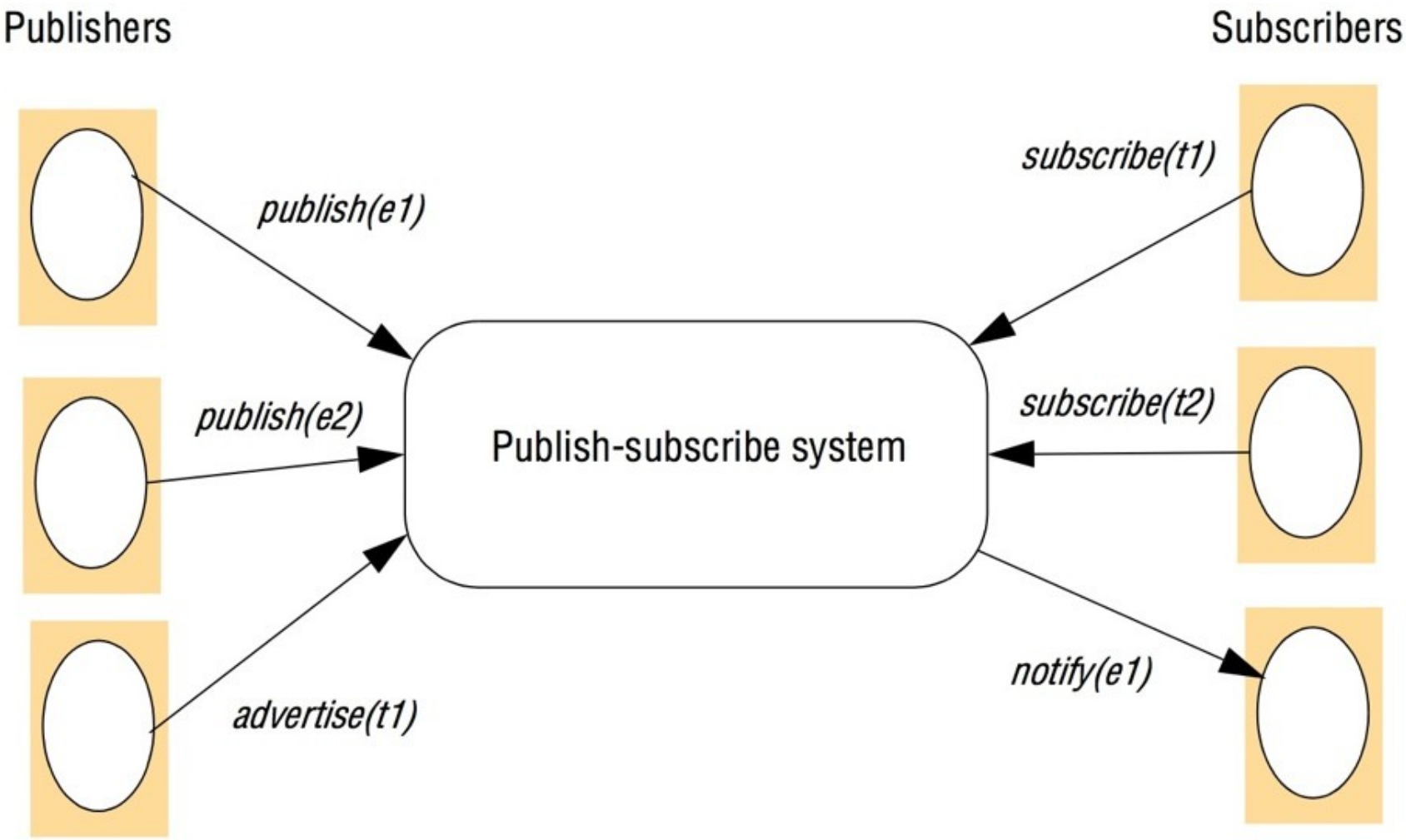
Modelos de MOMs

- Ponto-a-Ponto (Point-to-point)
- O MOM encaminha a mensagem recebida para a fila receptor.
- O MOM mantém um repositório de mensagens e desacopla o envio e a recepção de mensagens (assíncrono).

Modelos de MOMs

- Publicar-Assinar (Publish-Subscribe)
- Cada mensagem é associada à um tópico
  - A mensagem é colocada na fila de cada receptor inscrito no tópico
- Receptores se inscrevem no tópico para receber eventos publicados neste tópico
- Quando o evento ocorre, todos os receptores que assinaram o tópico são notificados
- Apropriado para comunicações em grupo (multicast)
- Inscrições duráveis permitem que receptores recebam o evento mesmo estando offline quando o evento ocorreu

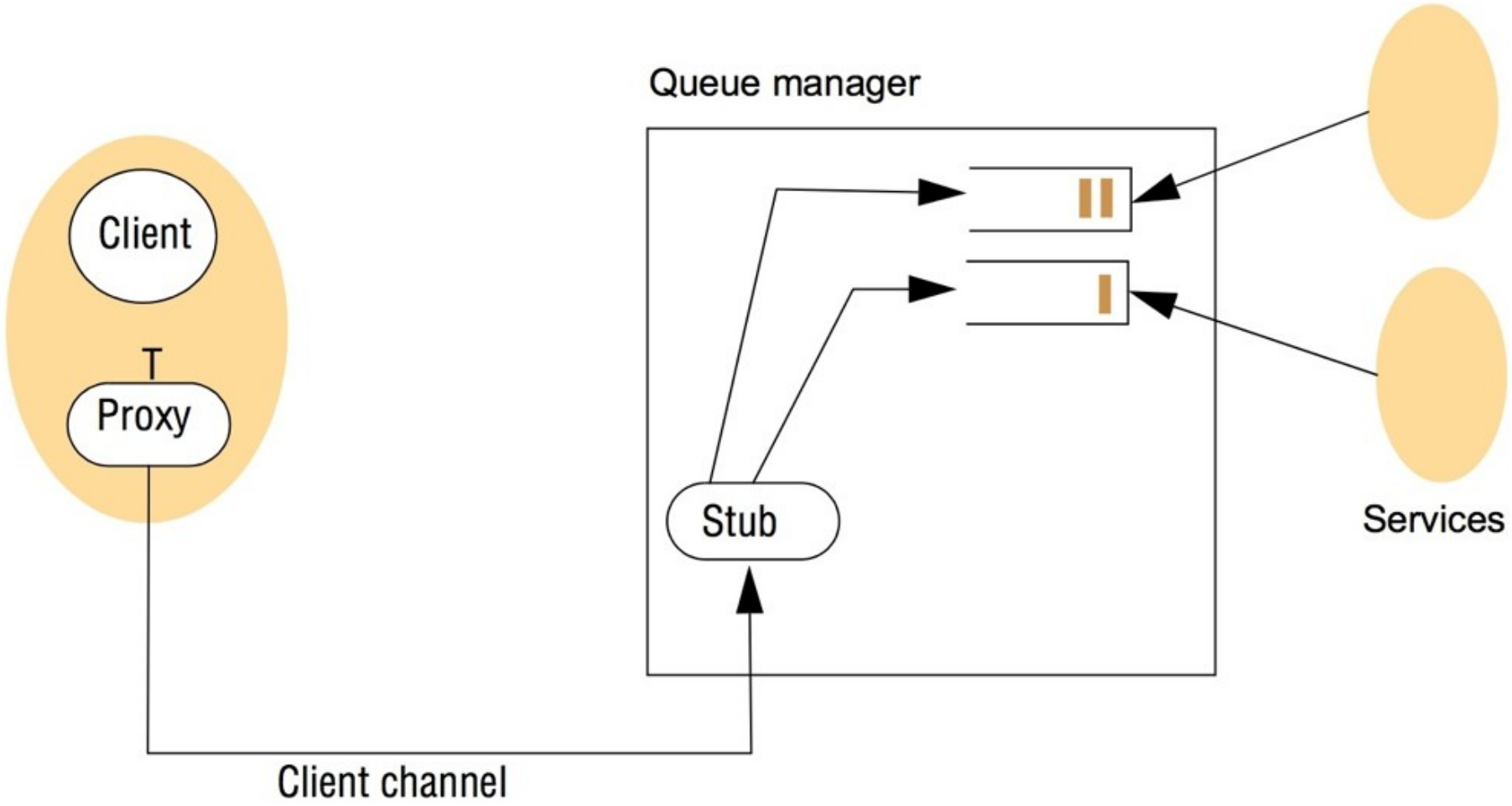
Sistema Publicar Assinar



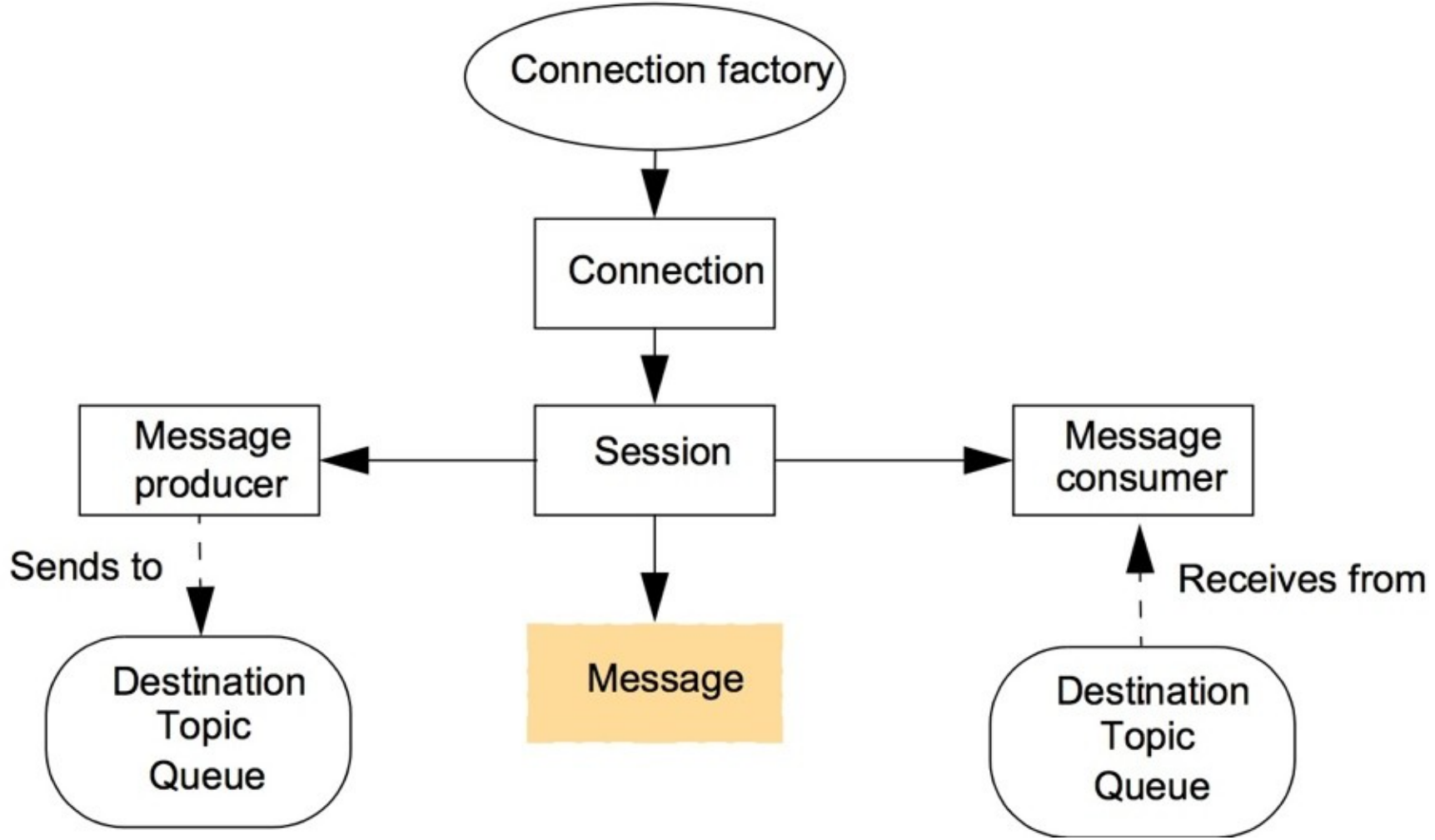
Ferramentas

- O MOM tem uma longa estória em aplicações distribuídas
  - Muito usado em mainframes
- Exemplos de ferramentas:
  - IBM MQ Series
  - Microsoft Message Queue
  - Rabbit MQ
  - Active MQ – usaremos nessa disciplina
  - Google Cloud Pub/Sub
  - AWS Simple Queue Service – SQS e AWS Message Queue - MQ
  - Azure Azure Queue Storage
  - APIs
  - Java Message Services JMS – usaremos como API para Active MQ
    - Permite conexão com qualquer produto que disponibilize um driver
    - Permite independência do provider

Arquitetura – IBM MQ Series



Arquitetura do Java Messaging Service



Dependências no Maven JMS usando AWS ActiveMQ

```
<dependency>
   <groupId>org.apache.activemq</groupId>
   <artifactId>activemq-client</artifactId>
   <version>5.15.0</version>
</dependency>

<dependency>
   <groupId>org.apache.activemq</groupId>
   <artifactId>activemq-pool</artifactId>
   <version>5.15.0</version>
</dependency>
```

Referência
https://docs.aws.amazon.com/pt_br/amazon-mq/latest/developer-guide/amazon-mq-working-java-example.html

Classe Java *ProdutorJMS*

```
package br.com.uniara.jms_test;

import javax.jms.*;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.jms.pool.PooledConnectionFactory;

public class ProdutorJMS {

        public static void main(String[] args) throws Exception {

            // Especificar parametros para conexao
            final String wireLevelEndpoint = "ssl://b-ef86b21c-5e3d-4b18-8a1b-286013517119-
1.mq.us-east-1.amazonaws.com:61617";
            final String activeMqUsername = "sdpc";
            final String activeMqPassword = "uniarauniara18";

            // Iniciar o Consumidor em um thread separado
            Thread t = new Thread(new ConsumidorJMS(wireLevelEndpoint,
activeMqUsername, activeMqPassword));
            t.start();
```

Classe Java *ProdutorJMS*

```
// Criar uma fabrica de conexoes
final ActiveMQConnectionFactory connectionFactory = new
ActiveMQConnectionFactory(wireLevelEndpoint);

// Informar username and password.
connectionFactory.setUserName(activeMqUsername);
connectionFactory.setPassword(activeMqPassword);

// Criar um Fabrica de Conexoes tipo Pooled para o produtor.
final PooledConnectionFactory pooledConnectionFactoryProducer = new
PooledConnectionFactory();
pooledConnectionFactoryProducer.setConnectionFactory(connectionFactory);
pooledConnectionFactoryProducer.setMaxConnections(10);

// Estabelecer uma conexao para o produtor
final Connection producerConnection =
pooledConnectionFactoryProducer.createConnection();
producerConnection.start();

// Criar uma session.
final Session producerSession = producerConnection.createSession(false,
Session.AUTO_ACKNOWLEDGE);
```

Classe Java *ProdutorJMS*

```
        // Criar uma fila chamada "MinhaFila".
        final Destination producerDestination =
producerSession.createQueue("MinhaFila");

        // Criar um produtor para a fila
        final MessageProducer producer =
producerSession.createProducer(producerDestination);
        producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

        // Enviar mensagens
        String text;
        for (int i = 0; i < 10; i++) {
            text = "Teste " + i;
            final TextMessage producerMessage =
producerSession.createTextMessage(text);

            // Envia a messagem
            producer.send(producerMessage);
            System.out.println("Mensagem enviada " + i);
        }
```

Classe Java *ProdutorJMS*

```
                    // Faz a "limpeza" dos recursos do produtor
                    producer.close();
                    producerSession.close();
                    producerConnection.close();

                    Thread.sleep(2000);
                    System.exit(0);
                }
}
```

Classe Java Consumidor*JMS*

```
package br.com.uniara.jms_test;

import javax.jms.*;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.jms.pool.PooledConnectionFactory;

public class ConsumidorJMS implements Runnable {

        // Parametros para conexao
        private String wireLevelEndpoint;
        private String activeMqUsername;
        private String activeMqPassword;

        public ConsumidorJMS(String url, String userName, String password) {
            this.wireLevelEndpoint = url;
            this.activeMqUsername = userName;
            this.activeMqPassword = password;
        }
```

Classe Java Consumidor*JMS*

```
public void run() {
    PooledConnectionFactory pooledConnectionFactoryConsumer = null;
    Connection consumerConnection = null;
    MessageConsumer consumer = null;
    Session consumerSession = null;

    try {

    // Criar uma fabrica de conexoes
    final ActiveMQConnectionFactory connectionFactory = new
ActiveMQConnectionFactory(wireLevelEndpoint);

    // Informar username and password.
    connectionFactory.setUserName(activeMqUsername);
    connectionFactory.setPassword(activeMqPassword);

    // Criar um Fabrica de Conexoes tipo Pooled para o consumidor
    pooledConnectionFactoryConsumer = new PooledConnectionFactory();
    pooledConnectionFactoryConsumer.setConnectionFactory(connectionFactory);
    pooledConnectionFactoryConsumer.setMaxConnections(10);
```

Classe Java Consumidor*JMS*

```
        // Estabelecer conexao para o consumidor
        consumerConnection = pooledConnectionFactoryConsumer.createConnection();
        consumerConnection.start();

        // Criar session.
        consumerSession = consumerConnection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

        // Criar uma fila chamada "MinhaFila".
        final Destination consumerDestination =
consumerSession.createQueue("MinhaFila");

        // Criar um consumidor para a fila.
        consumer = consumerSession.createConsumer(consumerDestination);


        int i = 0;
        while (i++ < 100) {
            // Aguardar por mensagens
            final Message consumerMessage = consumer.receive();

            // Receber mensagem quando chegar
            final TextMessage consumerTextMessage = (TextMessage) consumerMessage;
```

Classe Java Consumidor*JMS*

```
        } catch (JMSException ex) {
            System.err.println("Erro: " + ex.getMessage());
        } finally {
            // "Limpeza" dos recursos do consumidor
            if (consumer != null)
                try {
                    consumer.close();
                    if (consumerSession != null) consumerSession.close();
                    if (consumerConnection != null) consumerConnection.close();
                    if (pooledConnectionFactoryConsumer != null)
pooledConnectionFactoryConsumer.stop();
                } catch (JMSException e) {
                    e.printStackTrace();
                }

        }
    }
}
```

Trabalho - Chat usando JMS

- Considerando o exemplo dos slides anteriores, faça um programa de chat que permita dois ou mais usuários se comunicar em um chat online.
- O programa deve solicitor que o usuário digite seu nome ao iniciar a aplicação.
- O programa deve terminar quando o usuário digitar sair no console.
- Todas as mensagens digitadas pelos os usuários devem ser do formato:

```
29/03/2017 12:30 Rodrigo: Mensagem
```