Medii de proiectare și programare

2017-2018
Curs 11

Conținut

- Asynchronous messaging
- ActiveMQ -JMS, Spring JMS
- AQMP - RabbitMQ

Asynchronous messaging

- *Asynchronous messaging* este o modalitate de a trasmite indirect mesaje de la o aplicație la altă aplicație, fără a aștepta un răspuns.

- Este legată de comunicarea dintre aplicații.

- Diferă de alte mecanisme/modalități de comunicare prin modul în care informația este transmisă între sisteme.

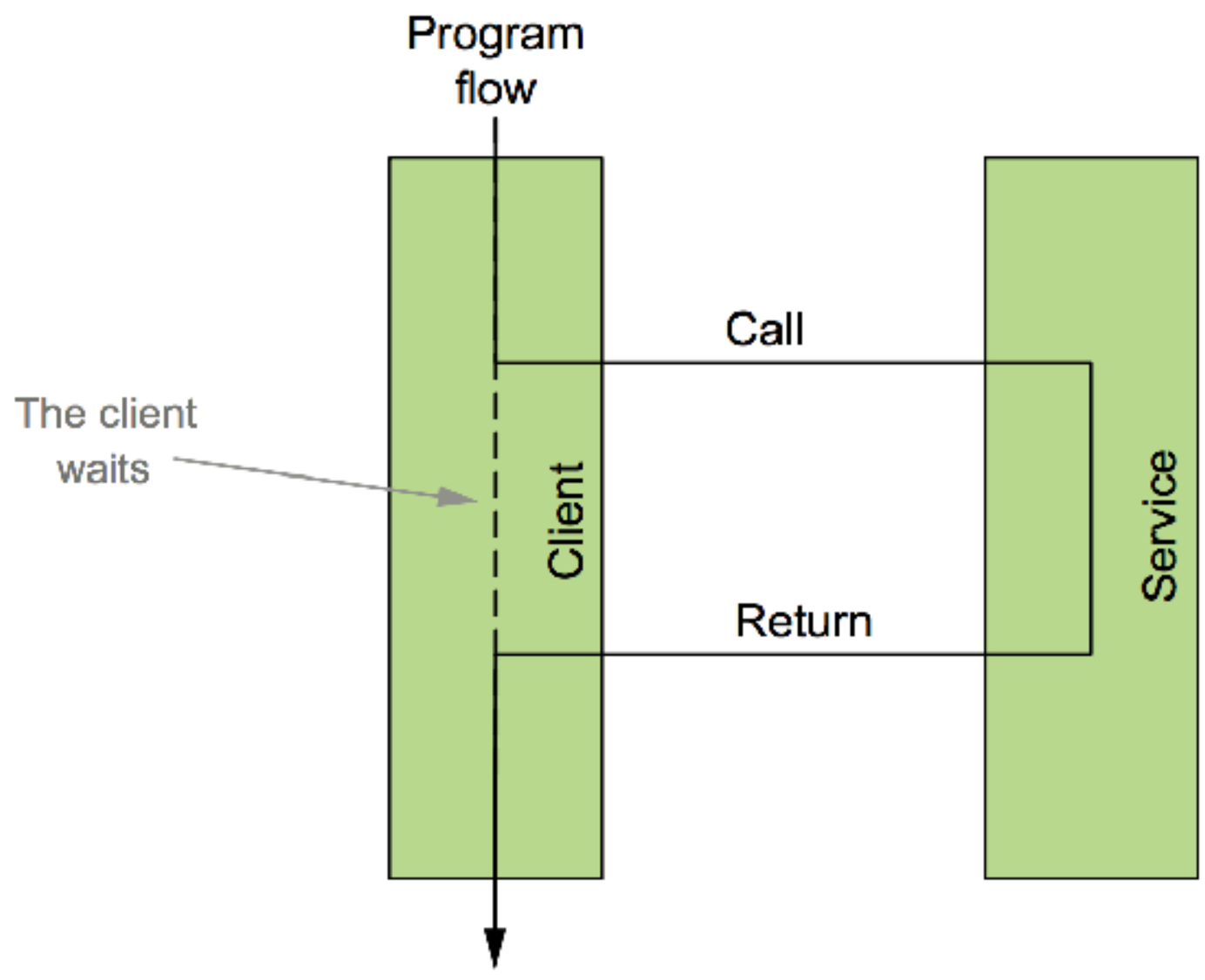
- Are câteva avantaje față de comunicarea sincronă (eng. *synchronous messaging*).

Comunicare sincronă

- Tehnologiile RPC ca RMI, .NET Remoting, Thrift folosesc comunicarea sincronă.
- Când un client apelează o metodă a unui obiect remote, clientul trebuie să aștepte terminarea execuției înainte de a trece mai departe.
- Chiar dacă metoda nu returnează

nimic, clientul tot trebuie să aștepte

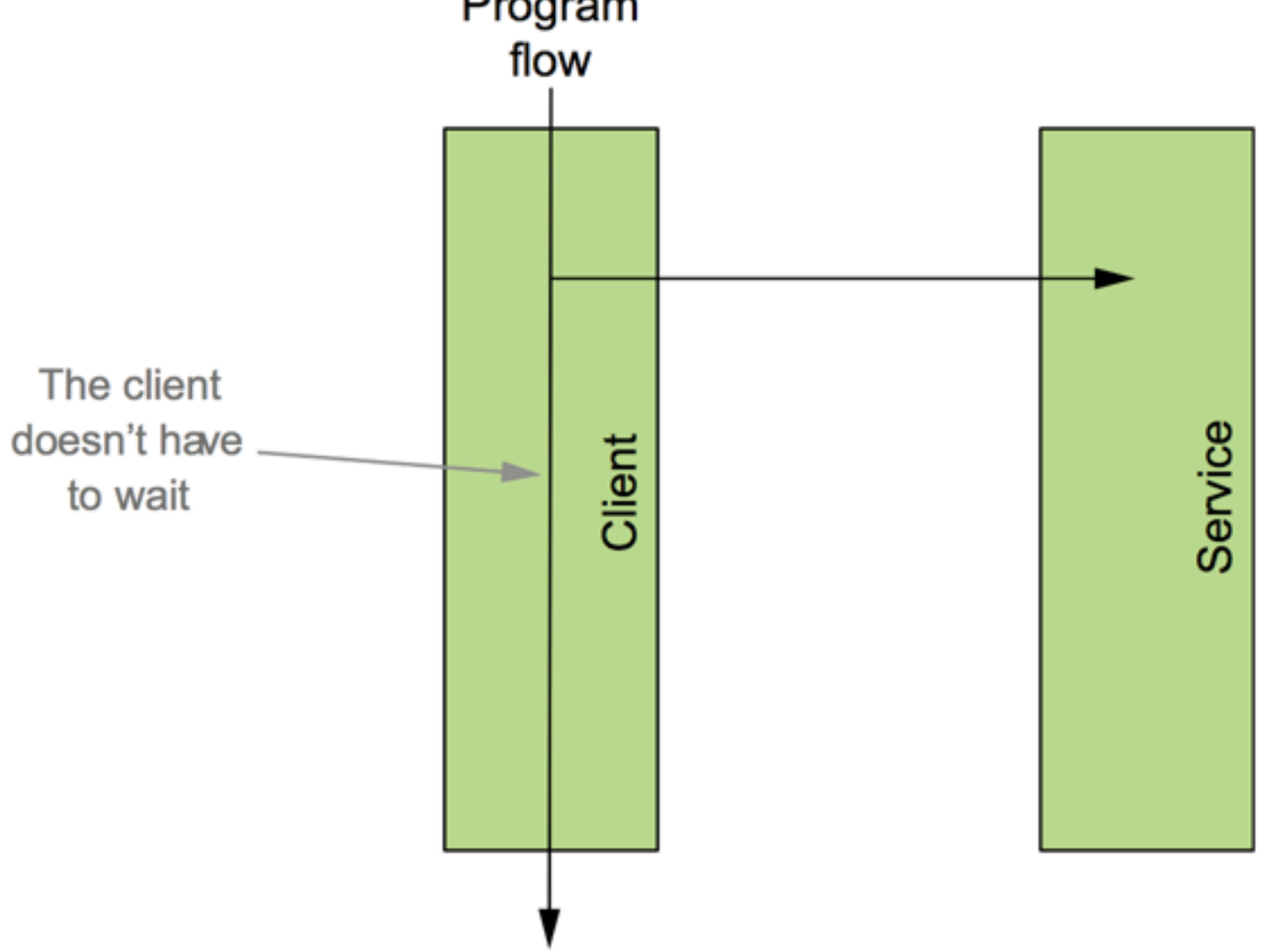
terminarea apelului.



Comunicarea asincronă

- Când mesajele sunt trimise asincron, clientul nu trebuie să aștepte terminarea procesării mesajului sau livrarea mesajului.
- Clientul trimite mesajul și trece la execuția următoarei instrucțiuni, presupunând ca serviciul va primi si va prelucra la un moment dat mesajul trimis.
- Acest tip de comunicare are câteva

  avantaje asupra comunicării sincrone.

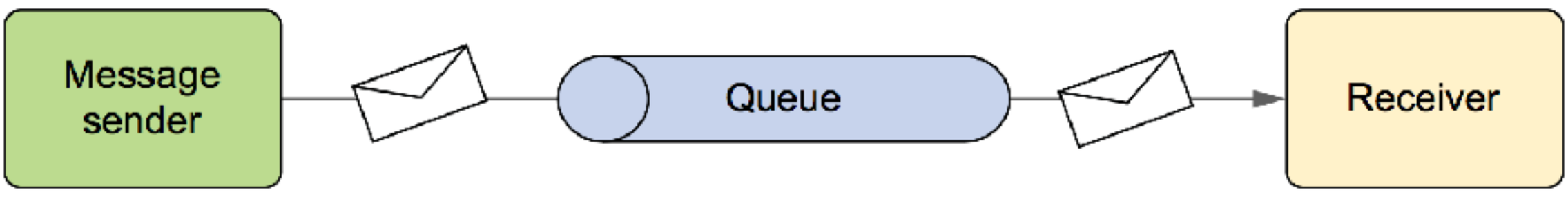


Asynchronous messaging

- Când o aplicație trimite un mesaj altei aplicații, nu există o legătura directă între cele două aplicații.
- Aplicația care trimite mesajul, îl trimite unui serviciu care asigură livrarea acestuia aplicației care îl așteaptă.
- Există doi actori principali: *brokerii de mesaje* și *destinațiile.*
- Când o aplicație trimite un mesaj, îl trimite unui broker de mesaje.
- Brokerul de mesaje asigură livrarea mesajului la destinația specificată, permițând expeditorului continuarea execuției.
- Mesajele trimise asincron au specificată o destinație.
- Destinația specifică doar locația de unde mesajele pot fi preluate, nu și cine le va prelua.
- Tipuri diferite de sisteme de trimitere a mesajelor asincrone pot folosi scheme diferite de rutare a acestora.
- Există două tipuri de bază de destinații: *cozi* și *topic-uri*.
- Fiecare tip de destinație are asociat un model specific de trimiterea a mesajelor:
  - *point-to-point* (corespunzător cozilor)
  - *publish/subscribe* (corespunzător topicurilor).

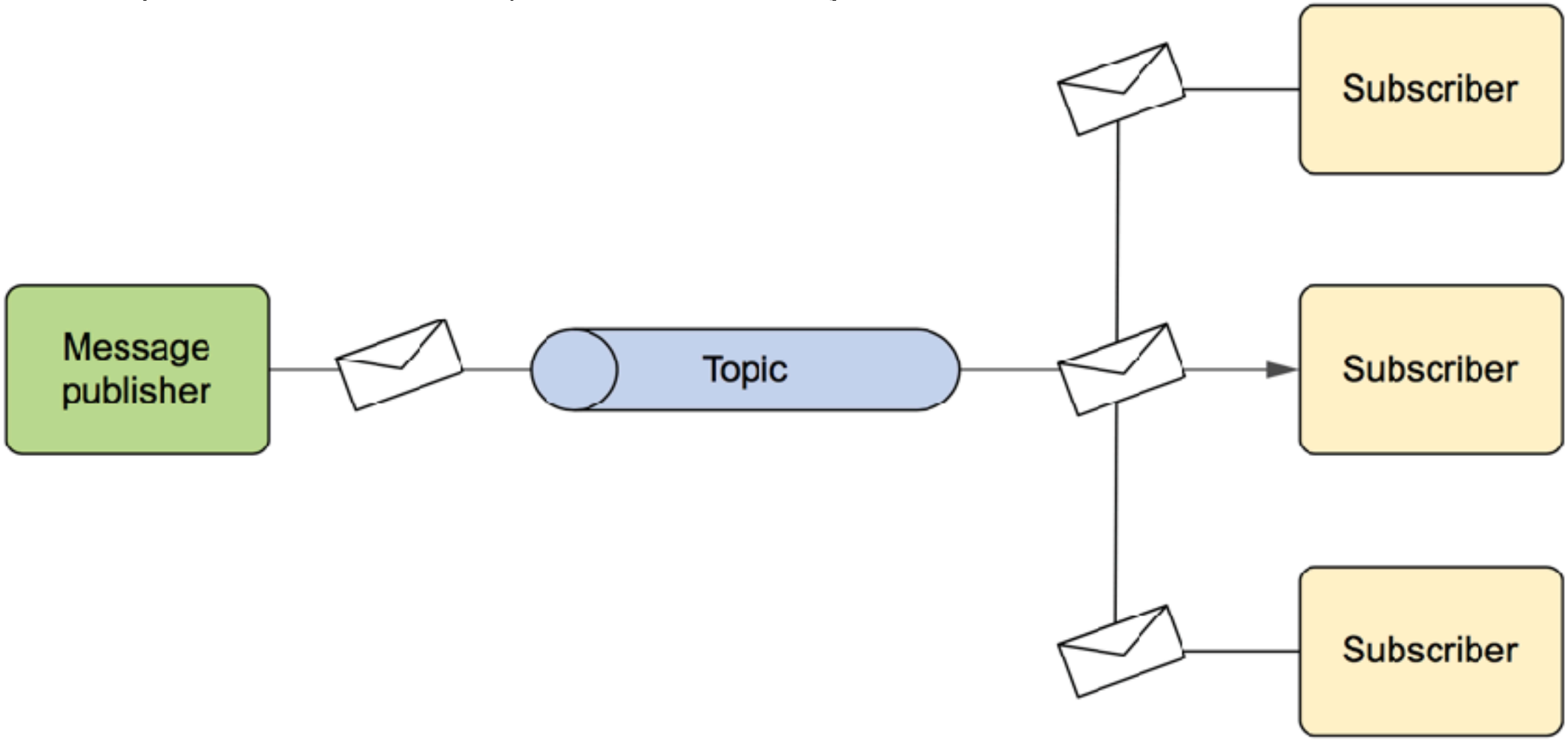
Point-To-Point Model

- În modelul point-to-point, fiecare mesaj are exact un expeditor și un destinatar. Când brokerul de mesaje primește un mesaj, pune mesajul într-o coadă. Când destinatarul cere următorul mesaj, mesajul este scos din coadă și trimis acestuia.
- Deoarece mesajul este șters din coadă în momentul trimiterii, se garantează livrarea acestuia unui singur destinatar.
- Chiar dacă fiecare mesaj din coada de mesaje este livrat unui singur destinatar, nu înseamnă că doar un singur destinatar scoate mesaje din coadă.
- Este posibil ca mai mulți destinatari să proceseze mesaje din coadă, dar fiecare va primi propriul mesaj.
- Dacă mai mulți destinatari așteaptă procesarea unui mesaj din coadă, nu se știe exact ce mesaj va procesa fiecare. Acest lucru permite aplicațiilor să mărească capacitatea de procesare a mesajelor prin adăugarea unui nou destinatar.



Publish-Subscribe Model

- In modelul publish/subscribe, mesajele sunt trimise unui *topic*. Asemănător cozilor, mai mulți destinatari pot aștepta livrarea unor mesaje de la un topic. Spre deosebire de cozi, când un mesaj este trimis unui topic toți destinatarii care așteaptă vor primi o copie a mesajului respectiv.
- Expeditorul (publisher) nu știe care sunt destinatarii (subscribers). El știe doar că mesajul va fi trimis unui topic, nu știe cine așteaptă mesaje la topicul respectiv.
- Expeditorul nu știe nici cum vor fi procesate mesajele.



Limitările comunicării sincrone

- Limitările clientului unui apel la distanță:
  - *Comunicarea sincronă implică așteptare.* Când un client apelează o metodă a unui obiect/serviciu remote, acesta trebuie să aștepte terminarea execuției acesteia înainte de a continua execuția. Dacă clientul face apeluri dese sau dacă apelul se execută mai lent, acest apel ar putea afecta performanța clientului.
  - *Clientul este cuplat cu serviciul prin interfața serviciului.* Dacă interfața serviciului se modifică, toți clienții acestuia trebuie să se modifice corespunzător.
  - *Clientul este dependent de locația serviciului.* Clientul trebuie configurat pentru a specifica locația serviciului (IP și port) pentru a ști unde este localizat serviciul. Dacă locația serviciului se modifică, clientul trebuie reconfigurat pentru a specifica noua locație.
  - *Clientul este dependent de disponibilitatea (eng. availability) serviciului.* Dacă serviciul devine indisponibil, clientul va eșua.
- Aceste limitări trebuie luate în considerare în momentul deciderii folosirii tipului de comunicare de mesaje potrivit aplicației.

Avantajele comunicării asincrone

- *Nu există timpi de așteptare*:
  - Când un mesaj este trimis asincron, clientul nu trebuie să aștepte livrarea sau procesarea mesajului. Clientul trimite mesajul brokerului de mesaje, cu încrederea că mesajul va ajunge la destinația specificată.
  - Deoarece nu trebuie să aștepte, clientul poate executa alte sarcini, îmbunătățind performanța acestuia.
- *Orientată pe mesaje și decuplată*
  - Spre deosebire de comunicarea RPC (orientată pe apelul unei metode), mesajele trimise asincron sunt centrate pe date. Clientul nu este dependent de o anumită signatura a metodei. Fiecare abonat la o coadă sau un topic care poate procesa datele trimise de client, poate procesa mesajul. Clientul nu trebuie să știe particularitățile specifice serviciului.
- *Independența locației*
  - Serviciile RPC sincrone sunt localizate de obicei prin adresa lor (IP și port), clienții fiind afectați de orice modificare a locației. Dacă adresa IP sau portul se modifică, clientul trebuie modificat corespunzător, altfel clientul nu va mai putea accesa serviciul.

Avantajele comunicării asincrone

- *Independența locației (cont.)*
  - In contrast, clienții sistemelor bazate pe mesaje nu știu ce serviciu va procesa mesajele lor sau locația acestora. Clientul știe doar de coada sau topicul prin intermediul căreia/căruia mesajul va fi transmis. Nu contează locația serviciului, atâta timp cât acesta poate procesa mesajele din coadă sau topic.
  - Servicii multiple se pot abona la un singur topic, fiecare primind o copie a aceluiași mesaj, dar fiecare poate procesa în mod diferit mesajul. Fiecare serviciu va folosi în mod independent aceeași informație primită de la topic.
  - În modelul point-to-point, se pot crea clustere de servicii. Dacă clientul nu este dependent de locația serviciului, și singura cerința legată de serviciu este că acesta trebuie să poată accesa brokerul de mesaje, mai multe servicii pot fi configurate să primească mesaje de la aceeași coadă. Dacă la un moment dat un serviciu devine suprasolicitat, se pot porni noi instanțe ale aceluiași serviciu care se abonează la aceeași coadă.

Avantajele comunicării asincrone

- Livrare garantată
  - Pentru ca un client să poată comunica cu un serviciu sincron, serviciul trebuie să fie disponibil la adresa și portul specificate. Dacă serviciul devine indisponibil (temporar sau pe o perioadă nedefinită), clientul nu își poate continua execuția.
  - Dacă mesajele sunt trimise asincron, clientul are garanția că mesajele sale vor fi livrate. Chiar dacă un serviciu nu este disponibil în momentul trimiterii mesajului, mesajul va fi păstrat de brokerul de mesaje până când serviciul va deveni din nou disponibil.

Brokeri de Mesaje

- **Apache ActiveMQ** (http://activemq.apache.org/)
  - Open source
  - Suportă clienți din limbaje și protocoale diferite: Java, C, C++, C#, Ruby, Perl, Python, PHP.
  - Oferă suport pentru frameworkul Spring, ActiveMQ putând fi ușor integrat și configurat în aplicații bazate pe Spring.
  - Suportă concepte avansate: grupuri de mesaje, destinații virtuale, destinații compuse
- **RabbitMQ** (https://www.rabbitmq.com/)
  - Open source message broker
  - Simplu și ușor de integrat și folosit (și in cloud).
  - Suportă mai multe protocoale de comunicare bazate pe mesaje.
  - Poate fi rulat pe diferite sisteme de operare și în cloud. Oferă instrumente de dezvoltare pentru diferite limbaje de programare (Java, .NET, PHP, Ruby, Python, etc).

ActiveMQ

- Instalarea:
  - Arhiva http://activemq.apache.org/
- Pornirea:
    - MacOS: *activemq start*
- Verificarea pornirii corecte:
    - http://127.0.0.1:8161/admin
    - Cont: user *admin*; pass *admin*
- Oprirea
    - MacOS: *activemq stop*

Java Message Service (JMS)

- Un standard Java care definește o interfață comună pentru lucrul cu brokeri de mesaje.
- Înainte de apariția JMS, fiecare broker de mesaje avea propriul API făcând codul aplicațiilor dependent de broker și greu de reutilizat pentru brokeri diferiți.
- Folosind JMS, toți brokerii de mesaje pot fi accesați folosind aceeași interfață comună (asemănător JDBC).

Trimiterea unui mesaj

```
ConnectionFactory cf = new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
        conn = cf.createConnection(); conn.start();
        session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Destination destination = new ActiveMQQueue(“test.queue");
        MessageProducer producer = session.createProducer(destination);
       TextMessage message = session.createTextMessage();
        message.setText("Hello world!");
        producer.send(message);  //trimiterea mesajului
 } catch (JMSException e) {
       // handle exception …
} finally {
       try {
           if (session != null) {   session.close();  }
           if (conn != null) { conn.close();  }
       } catch (JMSException ex) { //… tratarea exceptiilor
       }
}
```

Citirea unui mesaj

```
ConnectionFactory cf = new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
        conn = cf.createConnection();
        conn.start();
        session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Destination destination = new ActiveMQQueue("test.queue");
        MessageConsumer consumer = session.createConsumer(destination);
         Message message = consumer.receive();
         TextMessage textMessage = (TextMessage) message;
         System.out.println(“Primit mesaj: " + textMessage.getText());
 } catch (JMSException e) {
       // handle exception …
} finally {
       try {
            if (session != null) {   session.close();  }
            if (conn != null) { conn.close();  }
       } catch (JMSException ex) { //… tratarea exceptiilor
       }
}
```

Spring JMS

- Crearea unui *Connection Factory*
  - **`ActiveMQConnectionFactory`** este clasa JMS connection factory oferită de ActiveMQ

```
<bean id="connectionFactory"
    class="org.apache.activemq.spring.ActiveMQConnectionFactory" />
```

Implicit, **`ActiveMQConnectionFactory`** presupune că brokerul de mesaje așteaptă la locația *localhost* pe portul *61616*.

```
<bean id="connectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
     <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

Schimbarea locației implicite se poate face folosind proprietatea **`brokerURL`** brokerului de mesaje.

Spring JMS

- Declararea unei destinații pentru mesaje folosind ActiveMQ
  - Pentru trimiterea și livrarea mesajelor este necesară definirea unei destinații.
  - În funcție de tipul aplicației, destinația poate fi o coadă sau un topic.
  - Independent de tipul de destinație folosit, destinația trebuie specificată folosind clasa specifică corespunzatoare brokerului de mesaje folosit.
  - Declararea unei cozi

```
<bean id="queue" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="test.queue"/>
</bean>
```

  - Declararea unui topic

```
<bean id="topic" class="org.apache.activemq.command.ActiveMQTopic">
    <constructor-arg value="test.topic"/>
</bean>
```

  - Constructorul primește ca și parametru numele cozii de mesaje/topicului. Numele va fi folosit de brokerul de mesaje pentru trimiterea/livrarea mesajelor.
  - `test.queue`/`test.topic`.

Template-uri Spring JMS

- Clasa **`JmsTemplate`** este soluția oferită de Spring pentru a evita duplicarea codului JMS.
- **`JmsTemplate`** are ca și responsabilități crearea conexiunii, obținerea unei sesiuni, și trimiterea și livrarea mesajelor.
- Programatorii se ocupă doar de construirea mesajelor ce trebuie trimise, respectiv de procesarea mesajelor primite.
- **`JmsTemplate`** tratează toate excepțiile de tip **`JMSException (pachetul javax.jms)`** ce pot să apară și le convertește în propriile excepții de tip **`JMSException (pachetul org.springframework.jms)`**.
- Dacă o exceptie **`JMSException`** este aruncată pe perioada lucrului cu **`JmsTemplate`**, obiectul de tip **`JmsTemplate`** va prinde excepția și o va rearunca sub forma unei excepții specifice Spring (subclasă a excepției proprii **`JmsException`**).
- **`JMSException`** (din pachetul **`javax.jms`**) are o mulțime de subclase care descriu tipul de excepție apărut, dar toate sunt de tip checked.
- **`JmsTemplate`** prinde excepțiile checked și le rearuncă folosind excepții de tip runtime.

Template-uri Spring JMS

- Pentru a putea folosi obiecte de **`JmsTemplate`**, acestea trebuie declarate la configurare:

```
<bean id="jmsTemplate"
   class="org.springframework.jms.core.JmsTemplate">

    <constructor-arg ref="connectionFactory"/>

</bean>
```

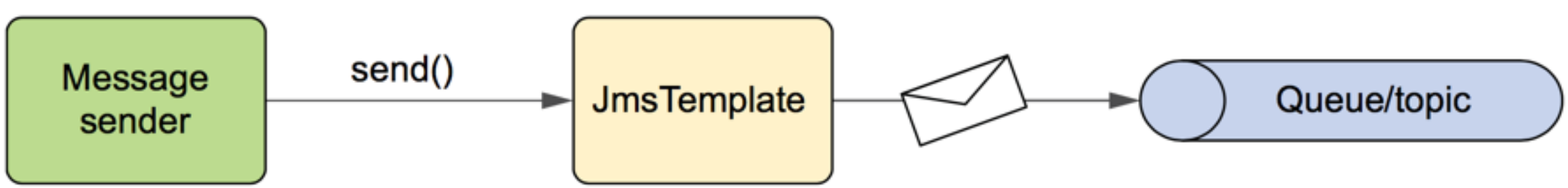
- Se pot seta și alte proprietăți.

Spring JMS - trimiterea unui mesaj

```
public interface AlertService {
  void sendAlert(Alert alert);
}
public class AlertServiceImpl implements AlertService {
  private JmsOperations jmsOperations; //interfata implementata de JmsTemplate
 public AlertServiceImpl(JmsOperations jmsOperatons) {
     this.jmsOperations = jmsOperations;
  }
  public void sendAlert(Alert alert) {
   //trimiterea unui mesaj
   jmsOperations.send( "alert.queue",            //Specificarea destinatiei
   new MessageCreator() {                       //Crearea mesajului
       public Message createMessage(Session session)
                throws JMSException {
          return session.createObjectMessage(alert);
       }
    } );
  }
}//AlertServiceImpl
```

Spring JMS - trimiterea unui mesaj

- Primul parametru al operației **`send()`** (interfața **`JmsOperations`**) specifică numele destinației (coadă sau topic).
- La apelul metodei **`send()`**, obiectul **`JmsTemplate`** se ocupă cu obținerea unei conexiuni și a unei sesiuni JMS și cu trimiterea mesajului din partea expeditorului.
- Mesajul este construit folosind un obiect de tip **`MessageCreator`**
- Metoda **`createMessage()`** (din clasa **`MessageCreator`**), cere sesiunii crearea unui mesaj trimitând ca și parametru obiectul **`Alert`** care păstrează datele conținute de mesajul transmis.
- Obiectul **`JmsTemplate`** tratează toate complexitățile trimiterii unui mesaj.



Spring JMS - trimiterea unui mesaj

- Specificarea unei destinații implicite:
  - De obicei, în cadrul unei aplicații mesajele se vor trimite la aceeași destinație.
  - Se poate specifica o destinație implicită în momentul instanțierii unui obiect de tip **`JmsTemplate`**.
  - Nu mai este necesară specificarea destinației în momentul trimiterii unui mesaj.

```
<bean id="jmsTemplate"
    class="org.springframework.jms.core.JmsTemplate">
     <constructor-arg ref="connectionFactory"/>
     <property name="defaultDestination" ref="queue"/>
</bean>

//Nu mai e necesara specificarea destinatiei
jmsOperations.send(
  new MessageCreator() {
  ...
  }
);
```

Spring JMS - trimiterea unui mesaj

- Convertirea mesajelor la trimitere
  - **`JmsTemplate`** oferă **`convertAndSend()`**. Spre deosebire de **`send()`**, metoda **`convertAndSend()`** nu primește ca și parametru un obiect de tip **`MessageCreator`**.
  - Metoda **`convertAndSend()`** folosește un obiect de tip message converter predefinit pentru a crea mesaje.

```
public void sendAlert(Alert alert) {
  jmsOperations.convertAndSend(alert);
}
```

  - **`JmsTemplate`** folosește o implementarea a interfeței **`MessageConverter`** pentru a converti obiectele în mesaje.
  - **`MessageConverter`** este o interfață definită de Spring având metodele:

```
public interface MessageConverter {
  Message toMessage(Object object, Session session)
                    throws JMSException, MessageConversionException;
  Object fromMessage(Message message)
                    throws JMSException, MessageConversionException;
}
```

MessageConverters

- Implicit **`JmsTemplate`** folosește un **`SimpleMessageConverter`** pentru trimiterea mesajelor folosind **`convertAndSend()`.**
- Se poate specifica alt tip de convertor la configurarea obiectului de tip JmsTemplate.

```
<bean id="converter"
  class="org.springframework.jms.support.converter.MappingJackson2MessageConverter">
      <property name="targetType" value="TEXT"/>
      <property name="typeIdPropertyName" value="_alert"/>
 </bean>
```

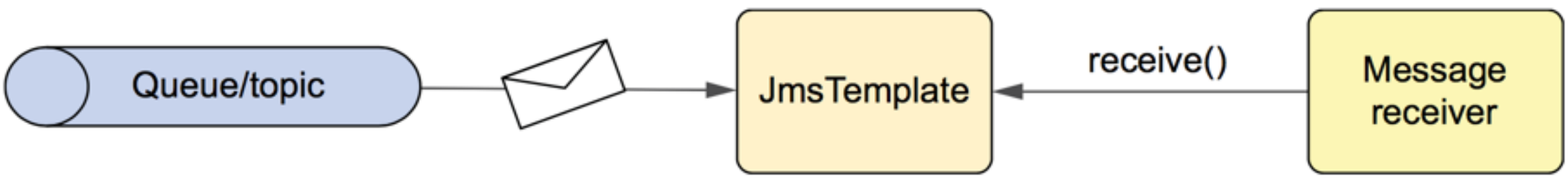
```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
      <constructor-arg ref="connectionFactory"/>
      <property name="defaultDestination" ref="topic"/>
      <property name="messageConverter" ref="converter"/>
</bean>
```

- **`MappingJackson2MessageConverter`**,
- **`MarshallingMessageConverter`** (JAXB XML format),
- **`SimpleMessageConverter`** (Strings to/from **`TextMessage`**, byte arrays to/ from **`BytesMessage`**, Maps to/from **`MapMessage`**, și Serializable objects to/from **`ObjectMessage`**)

Primirea/Procesarea mesajelor

- Apelul metodei **`receive()`** din **`JmsOperations`**.
- La apelul metodei **`receive()`**, se încearcă obținerea unui mesaj de la brokerul de mesaje. Dacă nici un mesaj nu este disponibil se așteaptă până când un mesaj este trimis la destinație.

```
public Alert receiveAlert() {
 try {
   ObjectMessage receivedMessage =(ObjectMessage)jmsOperations.receive();
   return (Alert) receivedMessage.getObject();
 } catch (JMSException jmsException) {
   throw JmsUtils.convertJmsAccessException(jmsException);
 }
}
```



Primirea/Procesarea mesajelor

- **`MessageConverters`** pot converti obiecte în mesaje la apelul metodei **`convertAndSend()`**.
- Pot fi folosiți și la procesarea mesajelor, prin apelul metodei **`receiveAndConvert()`** (**`JmsTemplate`**):

```
public Alert retrieveAlert() {
   return (Alert) jmsOperations.receiveAndConvert();
}
```

- Un dezavantaj al procesării mesajelor folosind metodele **`receive()`** și **`receiveAndConvert()`** este că ambele metode sunt sincrone.
- Apelantul metodelor trebuie să aștepte până când un mesaj devine disponibil (sau o condiție de timeout devine adevărată)

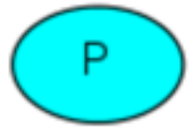
- Message driven beans (EJB) - pentru procesarea asincronă a mesajelor.

Advanced Message Queuing Protocol (AMQP)

- Protocol avansat pentru trimiterea mesajelor.
- În JMS există 3 tipuri de participanți:
  - expeditorul mesajului (producătorul),
  - destinatarul (destinatarii) mesajului (consumatorul/consumatorii),
  - canalul (coadă sau topic) folosit pentru livrarea mesajelor între expeditori și destinatari.
- In JMS, canalul decuplează expeditorul de destinatar, dar amândoi sunt cuplați de canal. Canalul are dublă responsabilitate:
  - primirea mesajelor
  - determinarea modului în care mesajele vor fi distribuite destinatarilor: coadă/ topic.

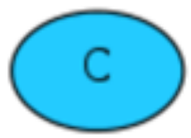
RabbitMQ - Terminologie

Producător (eng. *producer*): un program care trimite mesaje.



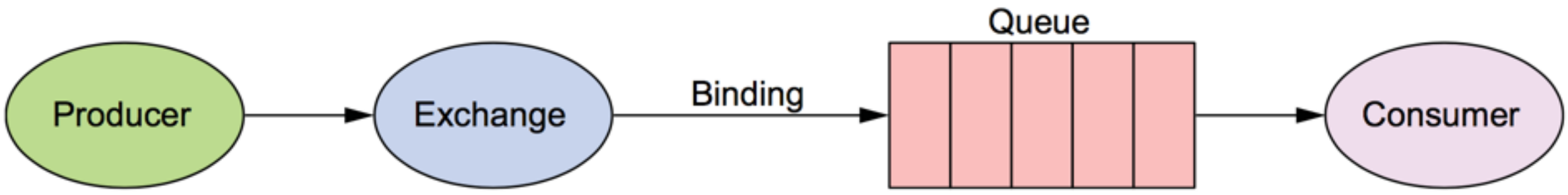
Coadă (eng. *queue*): o zonă tampon mare de mesaje, limitată doar de configurația mașinii pe care rulează RabbitMQ și dimensiunea HDD. Mai mulți producători pot trimite mesaje aceleiași cozi. Mai mulți consumatori pot citi/primi mesaje de la aceeași coadă.

Consumator (eng. *consumer*): un program care așteaptă să primească mesaje.



Advanced Message Queuing Protocol (AMQP)

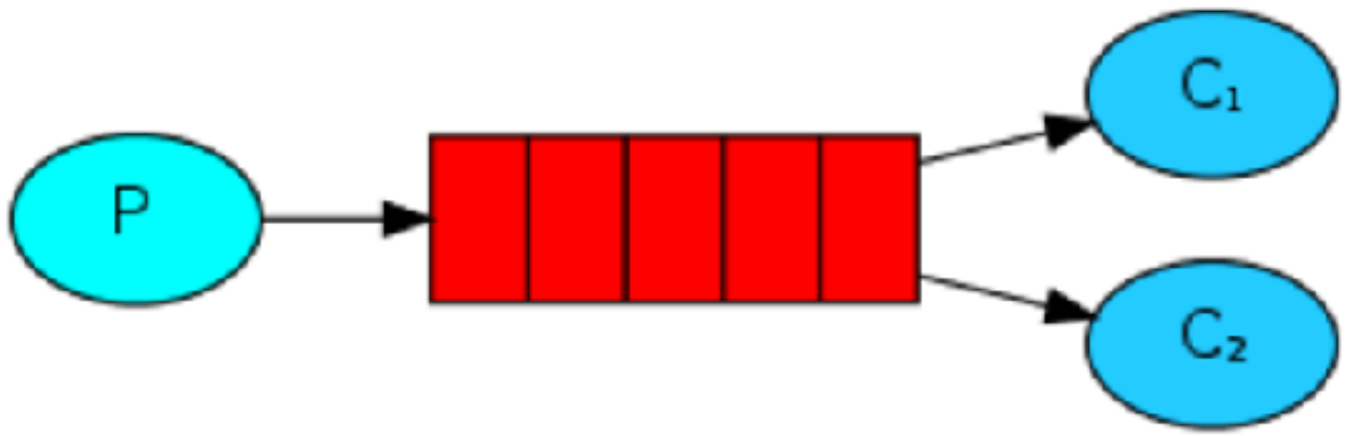
- Producătorii AMQP nu trimit mesajele direct unei cozi.
- AMQP introduce un nou nivel între producător și cozile care păstrează mesajele: *exchange*.
- Un producător de mesaje trimite mesajul unui exchange.
- Exchange-ul (care are asociate una sau mai multe cozi), rutează mesajele cozilor corespunzătoare.
- Consumatorii iau mesajele din cozi și le procesează. Algoritmul de rutare folosit are un impact minor asupra modului de dezvoltare a producătorilor/consumatorilor.
- Producătorii trimit mesajele unui exchange cu o cheie de rutare (eng. *routing key*), consumatorii iau mesajele din coada (adesea mesajele nu mai conțin cheia de rutare).
- RabbitMQ este un broker de mesaje bazat pe AMQP.



RabbitMQ - Modele asemănătoare JMS

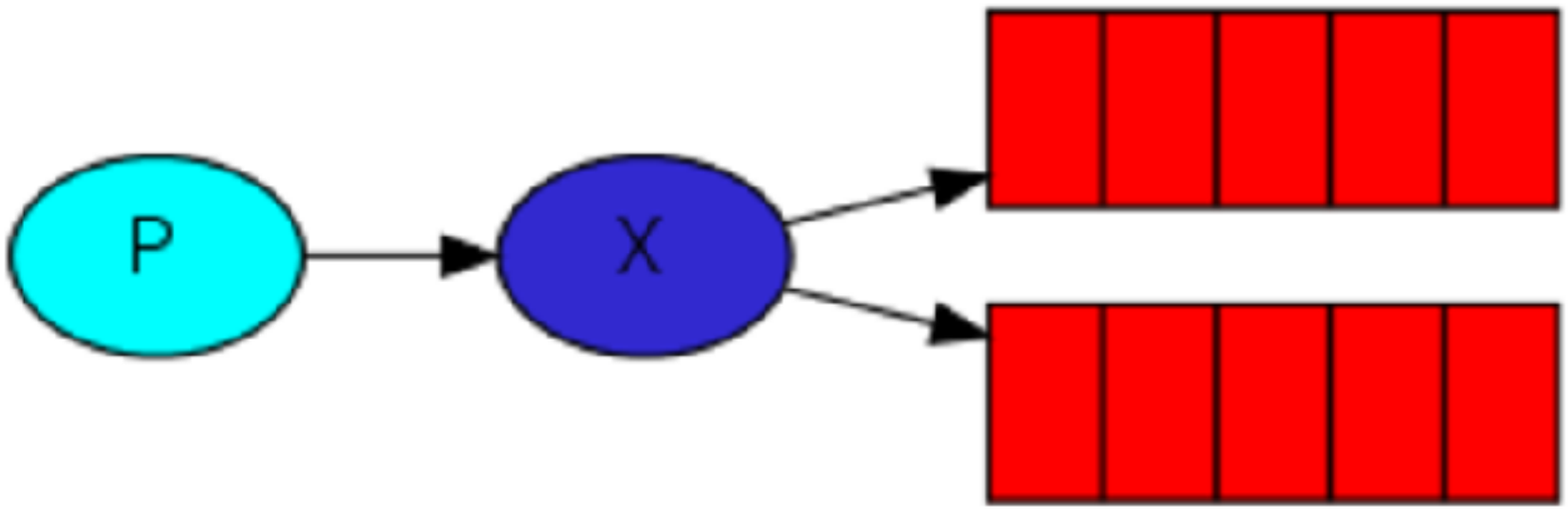
- Model simplu:

- Cozi de lucru:



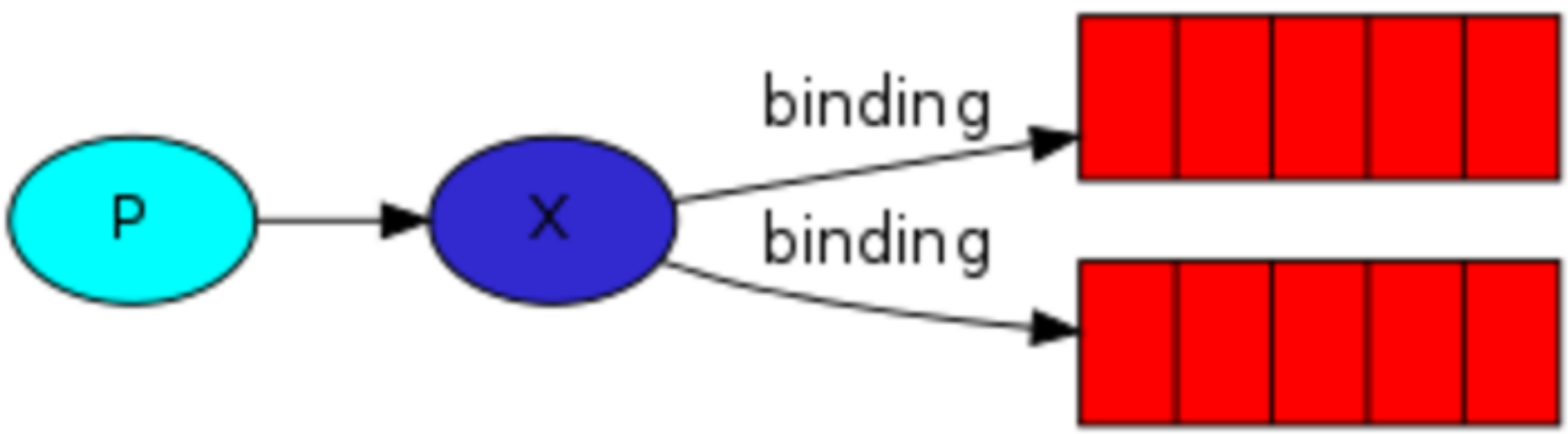
RabbitMQ - Model de transmitere a mesajelor

- Ideea de bază a modelului RabbitMQ este că producătorul nu trimite mesajul direct unei cozi.
- Producătorul nu știe nici dacă mesajul chiar va fi distribuit unei cozi.
- Producătorul poate doar să trimită mesaje unui *exchange*.
- Un *exchange* primește mesajele de la producători și le trimite spre cozile corespunzătoare.
- Un *exchange* trebuie să știe exact ce să facă cu un mesaj în momentul primirii:
  - Trebuie adăugat la o anumită coadă?
  - Trebuie adăugat la mai multe cozi?
  - Trebuie șters (eng. *discarded*)?
- Regulile folosite pentru luarea deciziei sunt definite de tipul de exchange:
  - *direct*,
  - *topic*,
  - *headers*
  - *fanout*.



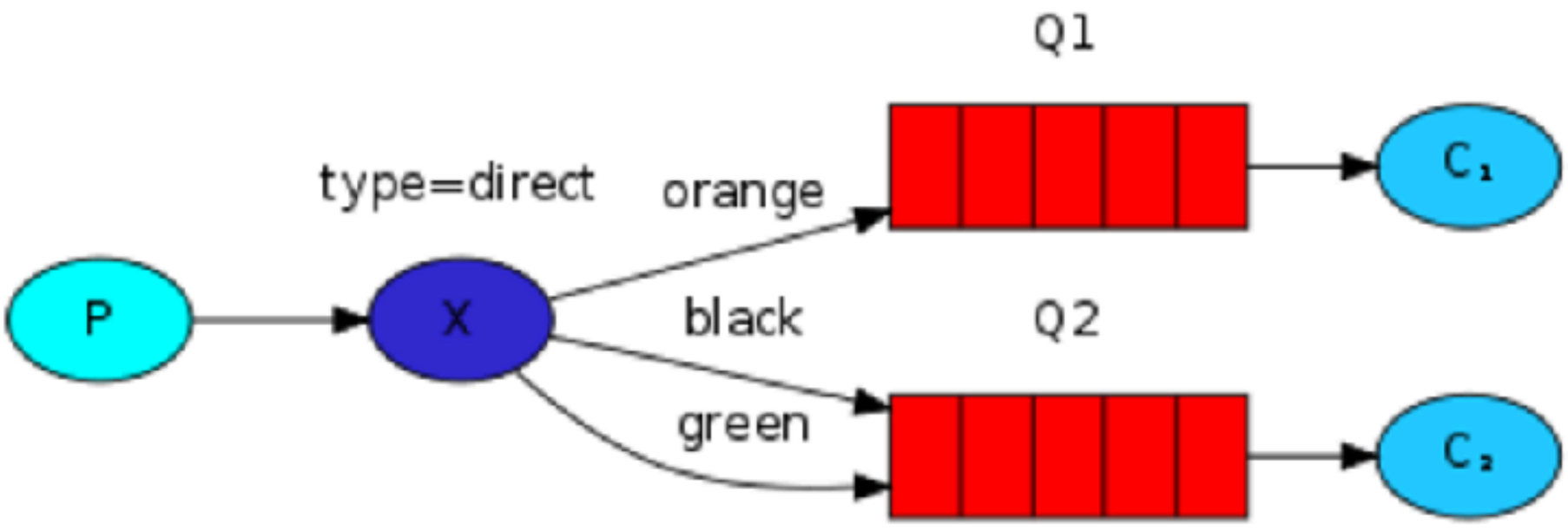
RabbitMQ - Fanout exchange

- *Fanout exchange* - distribuie mesajele tuturor cozilor cunoscute.
- Asocieri (eng. *bindings*): Relația dintre un exchange și o coadă este numită *asociere* (binding). Coada este interesată să primească mesaje de la exchange.
- Asocierile pot avea un extra parametru, numit cheie de asociere (eng. *binding key*)
- Semnificația cheii de asociere depinde de tipul de exchange.
- Fanout exchanges ignoră această valoare.



RabbitMQ - Direct exchange

- *Direct exchange*: - mesajul va fi distribuit cozilor care au asociate ca și chei exact cheia de rutare a mesajului.
- Exemplu:
  - direct exchange X cu două cozi asociate (Q1, Q2). Prima coadă are cheia de asociere *orange,* a doua coadă are asociate cheile: *black* și *green* .
  - Un mesaj cu cheia de rutare *orange* va fi distribuit cozii Q1.
  - Mesajele cu cheile de rutare *black* sau *green* vor fi distribuite cozii Q2.
  - Mesajele cu alte chei de rutare (ex. *white*, *red*, etc.) se vor pierde.

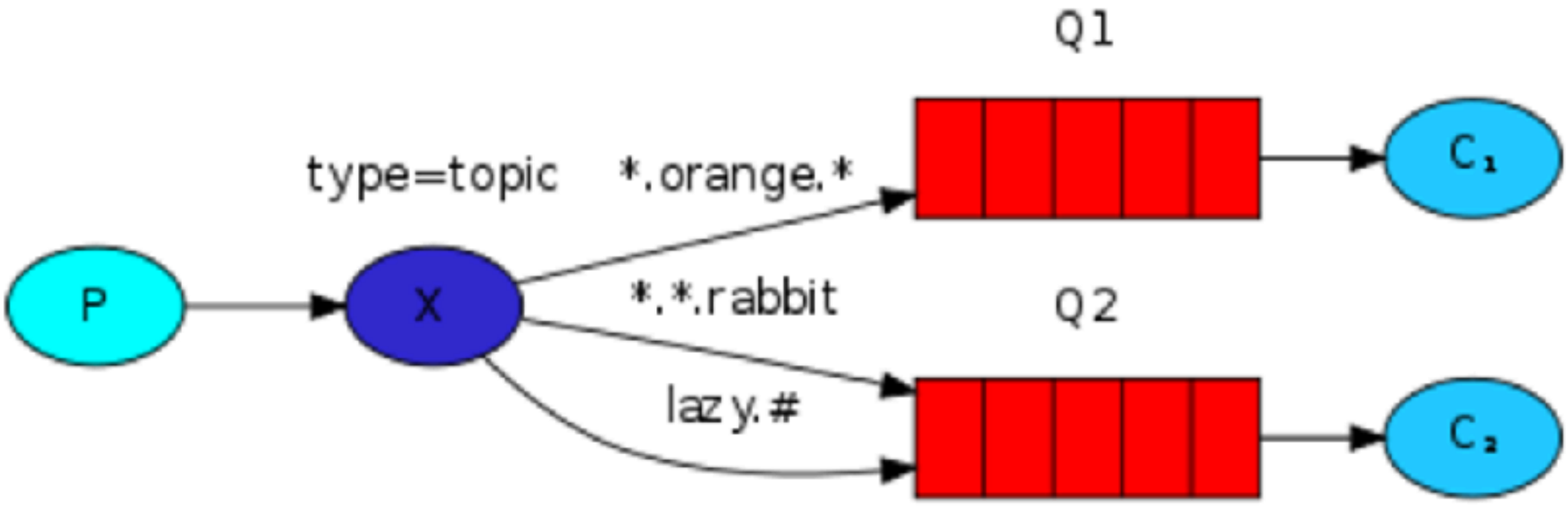


RabbitMQ - Topic exchange

- Mesajele trimise unui *topic exchange* nu pot avea o cheie de rutare arbitrară.
- Cheia de rutare trebuie să fie o listă de cuvinte separate prin punct. Cuvintele sunt de obicei caracteristici ale mesajelor.
- Exemple valide: "stock.usd.nyse", "nyse.vmw", "quick.orange.rabbit".
- In cheia de rutare pot fi oricâte cuvinte, dar lungimea să nu depășească 255 bytes.
- Cheia folosită pentru specificarea asocierilor are aceeași formă.
- Un mesaj având o anumită cheie de rutare va fi trimis tuturor cozilor asociate cu o cheie de asociere care se potrivește.
- Cazuri speciale pentru cheile folosite la specificarea asocierilor:
    - Caracterul * (star) poate înlocui un singur cuvânt.
    - Caracterul # (hash) poate înlocui zero sau mai multe cuvinte.
- Topic exchange se poate comporta ca și celelalte tipuri de exchange:
    - Când o coadă are asociată cheia "#" (hash) - va primi toate mesajele indiferent de cheia de rutare folosită (*fanout exchange*).
    - Dacă nu se folosesc caracterele speciale "*" și "#" la asocierile cozilor, topic exchange se va comporta ca și un *direct exchange*.

RabbitMQ -Exemplu (1)

- Trimiterea unor mesaje care descriu animale.
- Cheia de rutare este formată din 3 cuvinte: “<viteză>.<culoare>.<specie>".
  - S-au creat 3 asocieri (eng. *bindings*): coada Q1 este asociată cu cheia de asociere *“*.orange.**”*,* iar coada Q2 cu cheile *"*.*.rabbit"* și "lazy.#".
  - Sumar asocieri:
  - Q1 este interesată de toate animalele având culoarea *orange*.
  - Q2 este interesată de toți iepurii și de toate animalele lenețe.



RabbitMQ - Exemplu (2)

Cheia de rutare - Mesaj	Coada
quick.orange.rabbit	Q1, Q2
lazy.orange.elephant	Q1, Q2
quick.orange.fox	Q1
lazy.brown.fox	Q2
lazy.pink.rabbit	Q2 (o dată)
quick.brown.fox	Nici una
orange	Nici una
quick.orange.male.rabbit	Nici una
lazy.orange.male.rabbit	Q2

RabbitMQ

- *Headers Exchange*:
  - Este proiectat pentru rutarea folosind mai multe atribute, care sunt păstrate în antetul (header-ul) mesajului, și nu ca o cheie de rutare.
  - Cheia de rutare este ignorată. Atributele folosite pentru rutare sunt luate din antet.
  - Un mesaj este o *potrivire,* dacă valoarea din antet este egală cu valoarea precizată la asociere.
  - Este posibil să se facă asocierea dintre o coadă și un exchange folosind mai multe antete.
- În acest caz, broker-ul are nevoie de informații adiționale:
  - Ar trebui să ia în considerare mesajele care conțin cel puțin un antet sau toate antetele?
  - Se poate configura opțiunea folosind argumentul *x-match* de la binding:
    - *any* - este de ajuns ca un singur antet să apară
    - *all* - toate antetele trebuie să apară.

Referinte

- RabbitMQ:

  https://www.rabbitmq.com/getstarted.html

- Craig Walls, Spring in Action, Fourth Edition, Ed. Manning, 2015

- ActiveMQ:

http://activemq.apache.org/hello-world.html