UE Systèmes à objets et services répartis

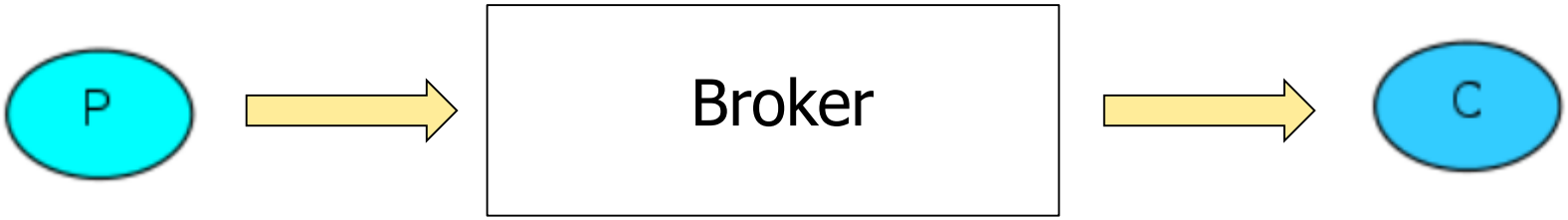
RabbitMQ : modèle publisher/subscriber

Frank Singhoff

Plan

1. **Concepts**
2. Queues
3. Exchanges
4. Conclusion

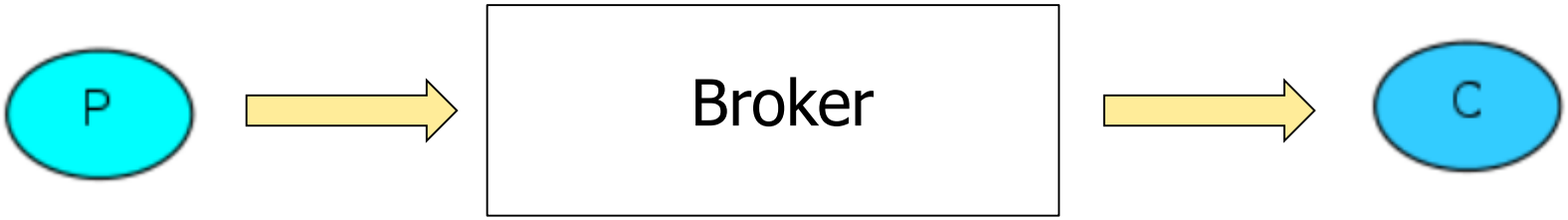
Rabbit MQ (Message queues)



- Middleware opensource (Mozilla Public Licence) développé par Pivotal Software
- https://github.com/rabbitmq/rabbitmq-server
- https://www.rabbitmq.com/

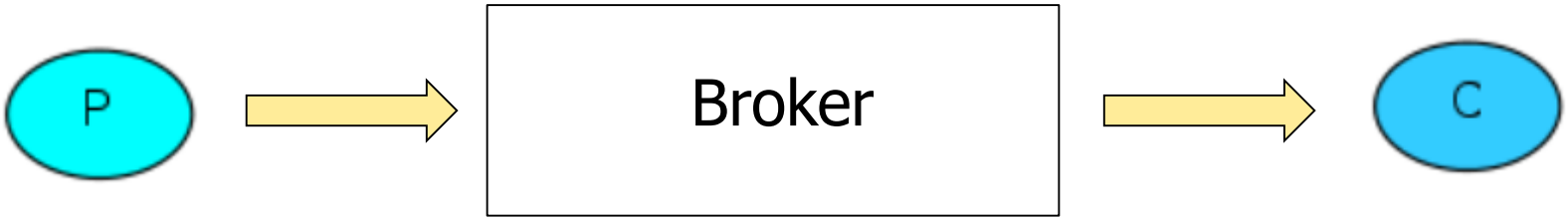
- Broker = serveur centralisé, notion de nœuds, de cluster
- Clients lourds pouvant être écrits en : Java, C#, Python, Objective-C, mais aussi PHP, Spring, Go, JavaScript, Elixir, Ruby
- Protocole AMQP (Advanced Message Queuing Protocol)

Rabbit MQ (Message queues)



- Producteurs (Publishers) : émission d'information
- Consommateurs (Subscribers) : réception d'information
- Serveur = Broker ; Client = Producteurs, consommateurs
- Modèles de communication:
  - Messages asynchrones.
  - Micro services.
  - Remote Procedure Call (type CORBA/Java RMI) … mais nettement moins sophistiqué

Rabbit MQ (Message queues)



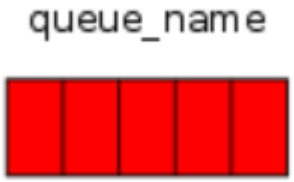
- Pas de transparence d'accès, à la localisation, à l'hétérogénéité
- Tolérance aux pannes : Messages persistances ou non, avec acquittement explicite ou non, gestion des connexions, clustering, …

- Couplage faible des publishers/producteurs et des subscribers/consommateurs
- Mécanisme de push coté subscribers (pull pour client/serveur).
- Subscribers, publishers et broker peuvent être localisés sur des machines différentes

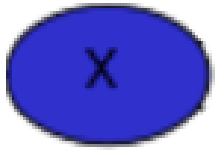
Concepts

- Producteur et Consommateur

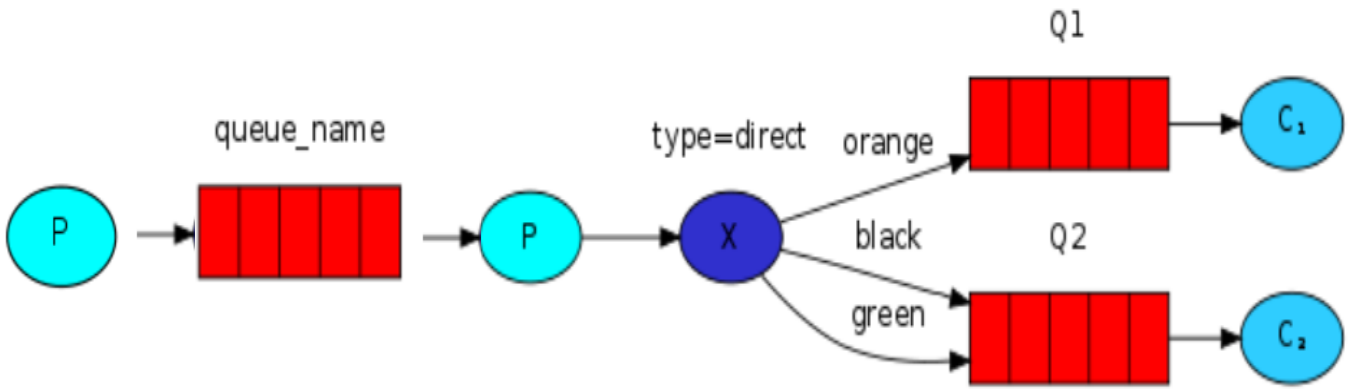
- Channel/connexion au serveur/nœuds. Connexion = application. Channel =thread/1 flux de message
- Message : transmis de façon asynchrone.
  - Acquittement explicite ou automatique
  - Durable ou non



- Queue : stockage des messages à consommer.
  - Persistante ou non.
- Exchange : traitements avant stockage dans les queues

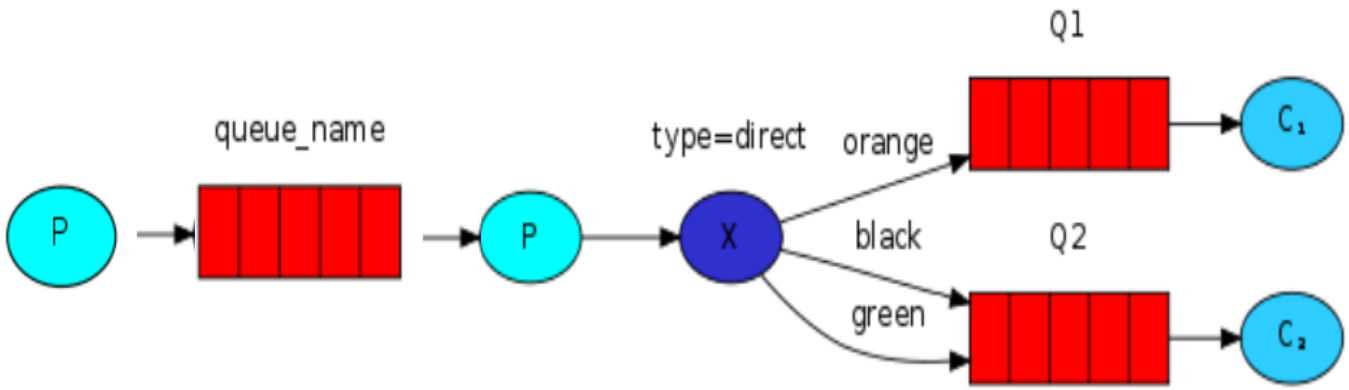


Concepts



- Binding : spécifie une liaison entre exchange et queues
- Routage : décrit comment les messages doivent être transmis du producteurs aux consommateurs. Mise en œuvre de filtrages ou de traitements (implantation de workflows).

Concepts

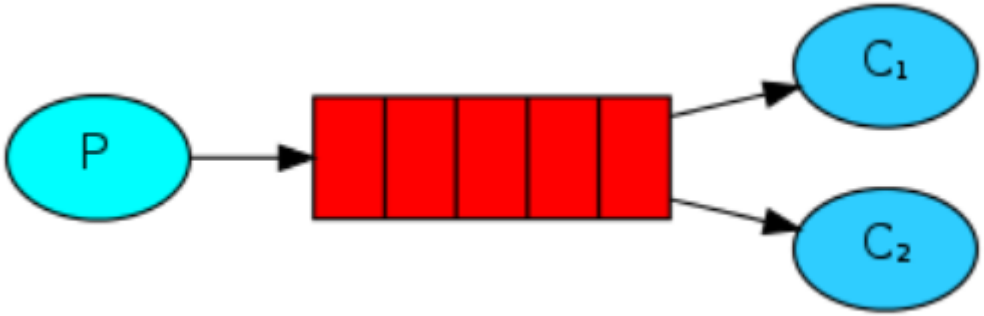


- Micro services
  - Découpage du logiciel en programmes élémentaires combinables
  - Micro service = programme élémentaire
  - « Separation of concern » : communication, filtrage, traitement
  - Réutilisation et combinaison des micro services
    - Ex : le même code pour consommer des messages de types différents.
  - Adapté à la tolérance aux pannes (disponibilité), au passage à l'échelle pour les performances (nombre d'instance d'un micro-service)

Plan

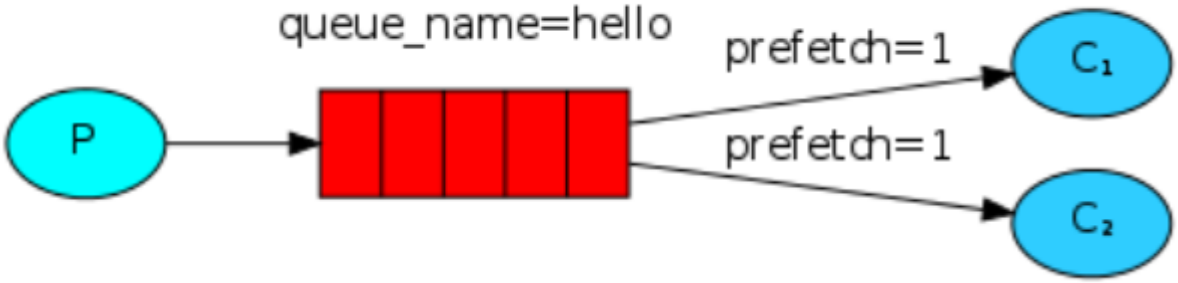
1. Concepts
2. **Queues**
3. Exchanges
4. Conclusion

Protocole de réception des messages



- Connection directe entre queues et producteurs/consommateurs
  - Plusieurs consommateurs => mécanisme de Round Robin => serveur concurrent
  - Message acquitté explicitement ou implicitement
  - Couplage plus fort, filtrage des messages à faire dans les clients

Protocole de réception des messages



- Lecture message quand consommateur est prêt
- Pas de lecture si pas d'acquittement (explicite ou implicite) du message précédent
- Mécanisme de tolérance aux pannes : message non retiré = non traité
- QoS = Quality of Service
- Qos/prefetch=1 : Réceptions multiples avant acquittement

Connection/Channel : communication avec le serveur RabbitMQ

- **ConnectionFactory :** paramétrage des connexions au serveur RabbitMQ : User login/passwd, paramètres TCP, …

- **Connection :** instancier une connexion AMQP vers le broker. Threadsafe.
- **Channel :** canal de communication vers le broker <u>pour UN thread (non Threadsafe)</u>

- **Exemple :**

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("127.0.0.1");  factory.setPort(8050);
factory.setUsername("user");  factory.setPassword("bitnami");

Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
```

Queues

- queueDeclare()

Actively declare a server-named exclusive, autodelete, non-durable queue.

- queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete, Map<String,Object> arguments)

Declare a queue

- queueBind(String queue, String exchange, String routingKey)

Bind a queue to an exchange, with no extra arguments.

- queueDelete(String queue)

Delete a queue, without regard it it is in use or has messages on it

- queuePurge(String queue)

Purges the contents of the given queue

Queues

- server-named – name chosen by the serveur
- durable - true if we are declaring a durable queue (the queue will survive a server restart)
- exclusive - true if we are declaring an exclusive queue (restricted to this connection)
- autoDelete - true if we are declaring an autodelete queue (server will delete it when no longer in use)

- ifUnused - true if the queue should be deleted only if not in use
- ifEmpty - true if the queue should be deleted only if empty

Emission, réception, acquittement de messages

- basicPublish(String exchange, String routingKey, AMQP.BasicProperties props, byte[] body)

Publish a message.

- basicConsume(String queue, boolean autoAck, String consumerTag, boolean noLocal, boolean exclusive, Map<String,Object> arguments, Consumer callback)

Start a consumer. Messages are read by a callback.

- basicAck(long deliveryTag, boolean multiple)

Acknowledge one or several received messages.

- basicNack(long deliveryTag, boolean multiple, boolean requeue)

Reject one or several received messages

- basicReject(long deliveryTag, boolean requeue)

Reject a message

Emission, reception, acquittement de messages

- autoAck - true if the server should consider messages acknowledged once delivered; false if the server should expect explicit acknowledgements
- consumerTag/deliveryTag - a client-generated tag to establish context
- noLocal - True if the server should not deliver to this consumer messages published on this channel's connection. Note that the RabbitMQ server does not support this flag.
- exclusive - true if this is an exclusive consumer

- multiple - true to acknowledge all messages up to and including the supplied delivery tag; false to acknowledge just the supplied delivery tag.
- requeue - true if the rejected message(s) should be requeued rather than discarded/dead-lettered

Exemple : producteur avec une queue

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("127.0.0.1");  factory.setPort(8050);
factory.setUsername("user");  factory.setPassword("bitnami");

Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
channel.queueDeclare ("hello", false, false, false, null);

System.out.println(" Producteur envoie : " + message);
channel.basicPublish("", "hello", null,
                    message.getBytes(StandardCharsets.UTF_8));
```

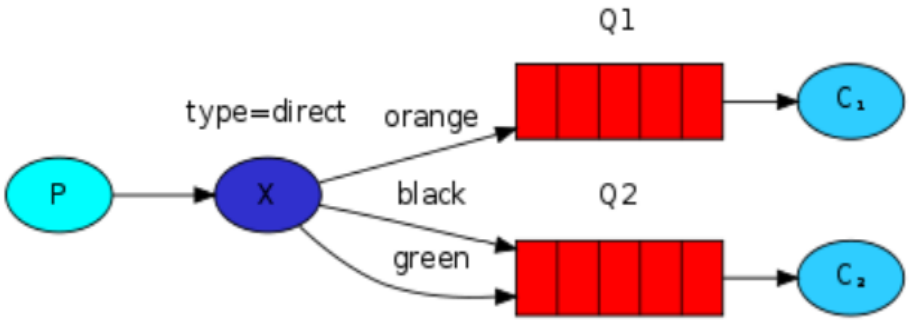
Exemple : consommateur avec une queue

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("127.0.0.1");  factory.setPort(8050);
factory.setUsername("user");  factory.setPassword("bitnami");

Connection connection = factory.newConnection();
Channel channel = connection.createChannel();

channel.queueDeclare("hello" , false, false, false, null);

System.out.println(" Récepteur : Attente de messages");
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
        String message = new String(delivery.getBody(), StandardCharsets.UTF_8);
        System.out.println("Réception du message=" + message);
};

channel.basicConsume(("hello", true, deliverCallback, consumerTag -> { });
```

Plan

1. Concepts
2. Queues
3. **Exchanges**
4. Conclusion

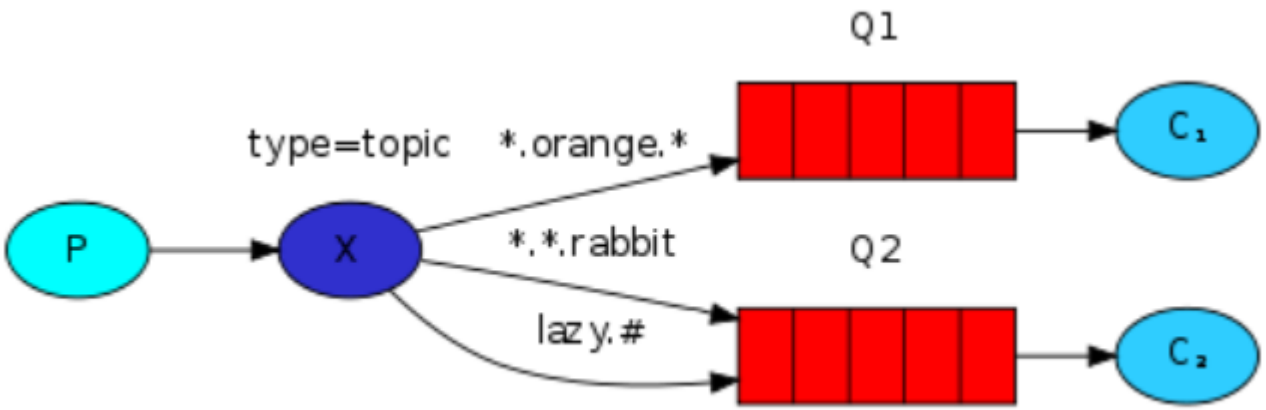
Exchanges



- Filtrage, Broadcast lors de la communication
- Principaux exchange : « direct » ou « topic »
- Pourquoi implanter le filtrage dans les exchanges plutôt que dans les clients :
  - « Separation of concern » (on consomme/produit ou on route)
  - Réutilisation/simplification du code des clients/consommateurs: changer les filtrages sans modifier le code des clients. Facilite également déploiement.
  - Architecture micro-services
  - Rappel : un client peut être producteur et consommateur

    => Flot de données comportant plusieurs queues successives

Exchanges



- Exchange « topic »
  - Routage sur plusieurs critères par des « topics »
  - Filtrage à partir d'expressions :
    - Mots séparés par des points. Ex : « caractère.couleur.espèce »
    - * : substitution par un mot
    - # : substitution par n'importe quel nombre de mots

Exchanges

- exchangeDeclare(String exchange, BuiltinExchangeType type)

Actively declare a non-autodelete, non-durable exchange

- exchangeDeclare(String exchange, BuiltinExchangeType type, boolean durable)

Actively declare a non-autodelete exchange

- exchangeDeclare(String exchange, String type)

Actively declare a non-autodelete, non-durable exchange

- exchangeDeclare(String exchange, String type, boolean durable)

Actively declare a non-autodelete exchange

- exchangeBind(String destination, String source, String routingKey)

Bind an exchange to an exchange

- exchangeUnbind(String destination, String source, String routingKey)

Unbind an exchange from an exchange

- exchangeDelete(String exchange)

Delete an exchange, without regard for whether it is in use or not

Exchanges

- durable - true if we are declaring a durable exchange (the exchange will survive a server restart)
- autoDelete - true if the server should delete the exchange when it is no longer in use
- internal - true if the exchange is internal, i.e. can't be directly published to by a client.

- BuiltinExchangeType **:** Direct, Fanout, Headers, Topic

Squelette d’un producteur avec exchange

1. Création d’un canal avec le serveur
2. Déclaration d’un exchange pour le canal
3. Publication d’un message sur le canal

Squelette d'un consommateur avec exchange

1. Création d'un canal avec le serveur

2. Déclaration d'un exchange pour le canal

3. Liaison (binding) des clefs/topics à la file d'attente associée à l'exchange

4. Enregistrement d'un callback de réception

Exemple : producteur avec un exchange

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("127.0.0.1");  factory.setPort(8050);
factory.setUsername("user");  factory.setPassword("bitnami");

channel.exchangeDeclare("mon_exchange_couleur", "topic");

String m1 = "n existe pas";
channel.basicPublish("mon_exchange_couleur", "elephant.rose.rapide",
      null, m1.getBytes("UTF-8"));

String m2 = "n existe plus";
channel.basicPublish("mon_exchange_couleur", "elephant.gris.lent",
      null, m2.getBytes("UTF-8"));
```

Exemple : consommateur avec un exchange

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("127.0.0.1");  factory.setPort(8050);
factory.setUsername("user");  factory.setPassword("bitnami");

channel.exchangeDeclare("mon_exchange_couleur", "topic");
String queueName = channel.queueDeclare().getQueue();

String bindingKey1="*.rose.*";
String bindingKey2="elephant.#";
channel.queueBind(queueName, "mon_exchange_couleur", bindingKey1);
channel.queueBind(queueName, "mon_exchange_couleur", bindingKey2);

DeliverCallback deliverCallback = (consumerTag, delivery) -> {
   String message = new String(delivery.getBody(), "UTF-8");
   System.out.println(" Message='" + delivery.getEnvelope().getRoutingKey() + "':'" + message + "'");
};

channel.basicConsume(queueName, true, deliverCallback, consumerTag -> { });
```

Plan

1. Concepts
2. Queues
3. Exchanges
4. **Conclusion**

Conclusion

- Ce cours comporte de nombreux éléments extraits de la documentation RabbitMQ disponible ici :
  https://www.rabbitmq.com/getstarted.html

- Echange de messages asynchrones
- Micro-services et bonnes propriétés concernant la conception du logiciel
- Concepts: Queues, binding/clef de routage, Exchange

- Conception conjointe du logiciel et du flot de données
- Pas de transparence ni à l'hétérogénéité, ni à l'accès, ni à la localisation, mais intégration de nombreux langages de programmation