MICROSERVICES

JÉRÉMY PERROUAULT

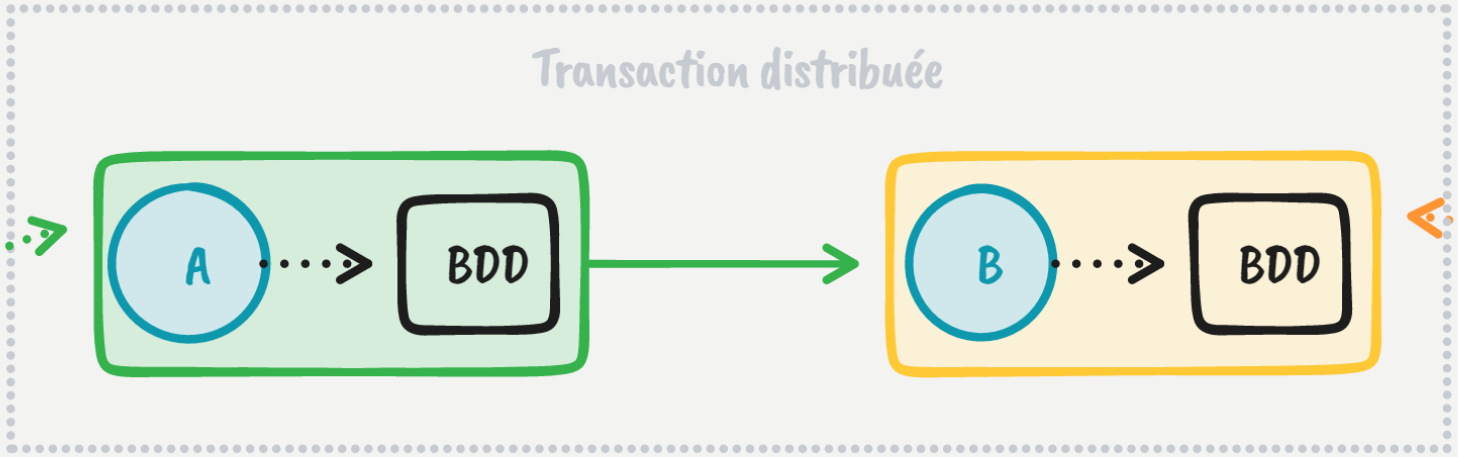
SAGA

GESTION DES TRANSACTIONS

SAGA

- Dans l'architecture implémentée, il y a un problème majeur
  - La (non) gestion d'une transaction n'est pas compensée

SAGA



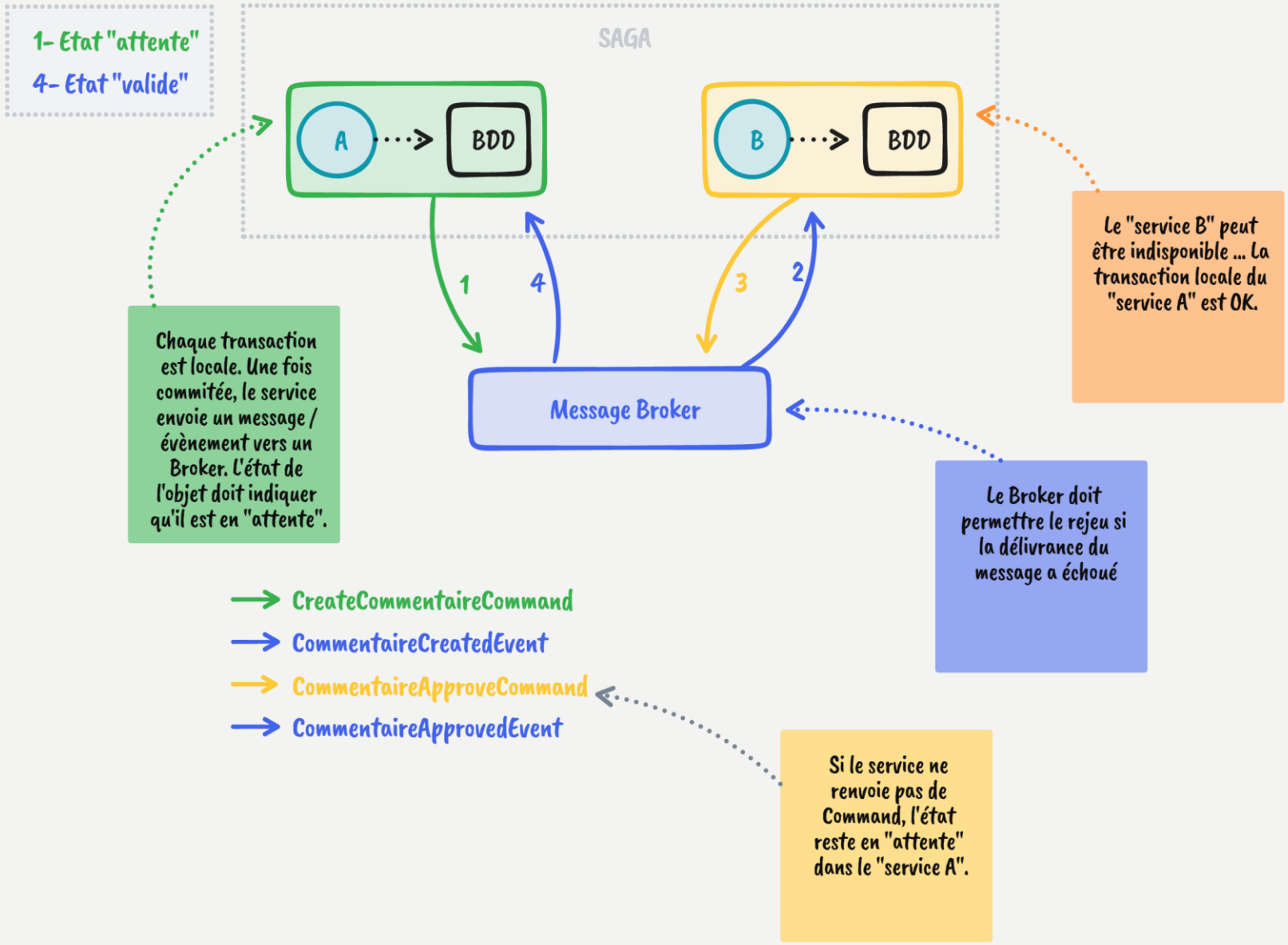
Une transaction démarre. Elle se commit lorsqu'on a reçu une réponse de "service B"

Le "service B" doit être disponible pour valider la transaction

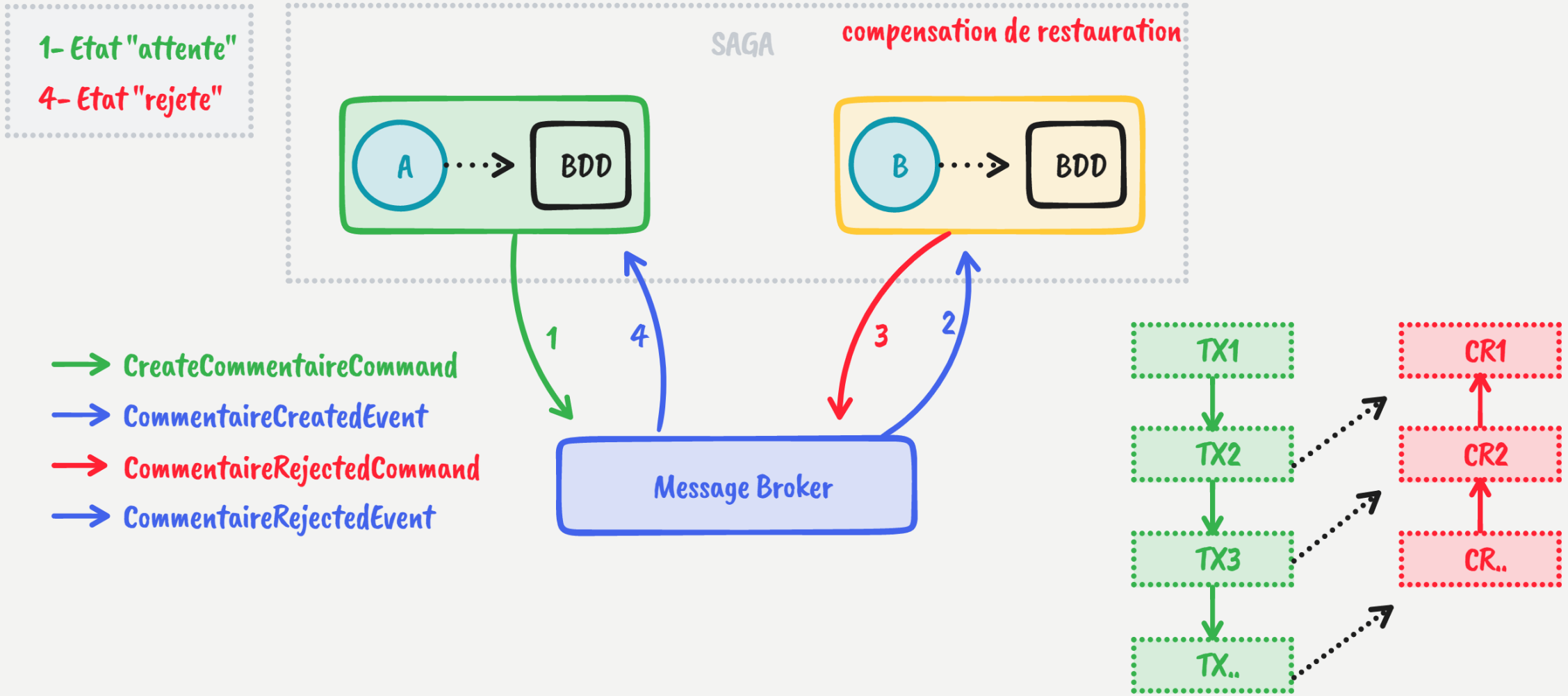
SAGA

- Utilisation du Pattern « Saga »
  - L'idée, c'est d'avoir une séquence de transactions locales
    - Chaque transaction met à jour les changements locaux
    - Puis publie un message ou un évènement pour que les autres services puissent se mettre à jour
    - Et en cas d'échec lors de la transaction locale, un autre message / évènement est publié
      - Chaque service impliqué pourra « revenir en arrière » en déclenchant son mécanisme de compensation de transaction

SAGA


1- Etat "attente"
4- Etat "valide"
SAGA
A
BDD
B
BDD
1
4
3
2
Message Broker
Chaque transaction est locale. Une fois commitée, le service envoie un message / évènement vers un Broker. L'état de l'objet doit indiquer qu'il est en "attente".
Le "service B" peut être indisponible ... La transaction locale du "service A" est OK.
Le Broker doit permettre le rejeu si la délivrance du message a échoué
CreateCommentaireCommand
CommentaireCreatedEvent
CommentaireApproveCommand
CommentaireApprovedEvent
Si le service ne renvoie pas de Command, l'état reste en "attente" dans le "service A".

SAGA


1- Etat "attente"
4- Etat "rejete"
SAGA
compensation de restauration
A
BDD
B
BDD
1
4
3
2
Message Broker
CreateCommentaireCommand
CommentaireCreatedEvent
CommentaireRejectedCommand
CommentaireRejectedEvent
TX1
TX2
TX3
TX..
CR1
CR2
CR..

SAGA

- Il y a deux façons d'implémenter Saga
  - Chorégraphie
    - Les services s'échangent des messages sans contrôle centralisé
  - Orchestration
    - Les services s'échangent des messages via un contrôleur centralisé, un orchestrateur

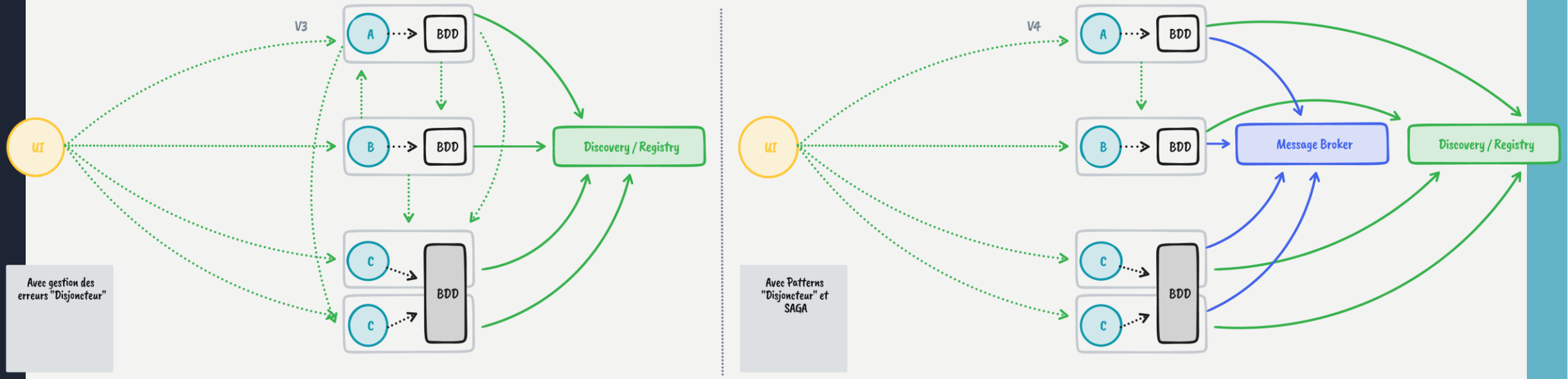
SAGA

- Utilisation d'une technologie Broker existante …
  - RabbitMQ
  - Kafka
  - Amazon SQS
  - Axon
  - EventStoreDb
  - Redis Stream
  - …

SAGA

- Attention, selon ce qu'on veut faire, on utilisera différentes implémentations
  - Messaging infrastructure (RabbitMQ)
  - Event Streaming / Storing / Processing (Kafka)

SAGA


V3
A
BDD
UI
B
BDD
Discovery / Registry
C
C
BDD
Avec gestion des erreurs "Disjoncteur"
V4
A
BDD
UI
B
BDD
Message Broker
Discovery / Registry
C
C
BDD
Avec Patterns "Disjoncteur" et SAGA

SAGA

- Mettre en place RabbitMQ
- Configurer les 2 services

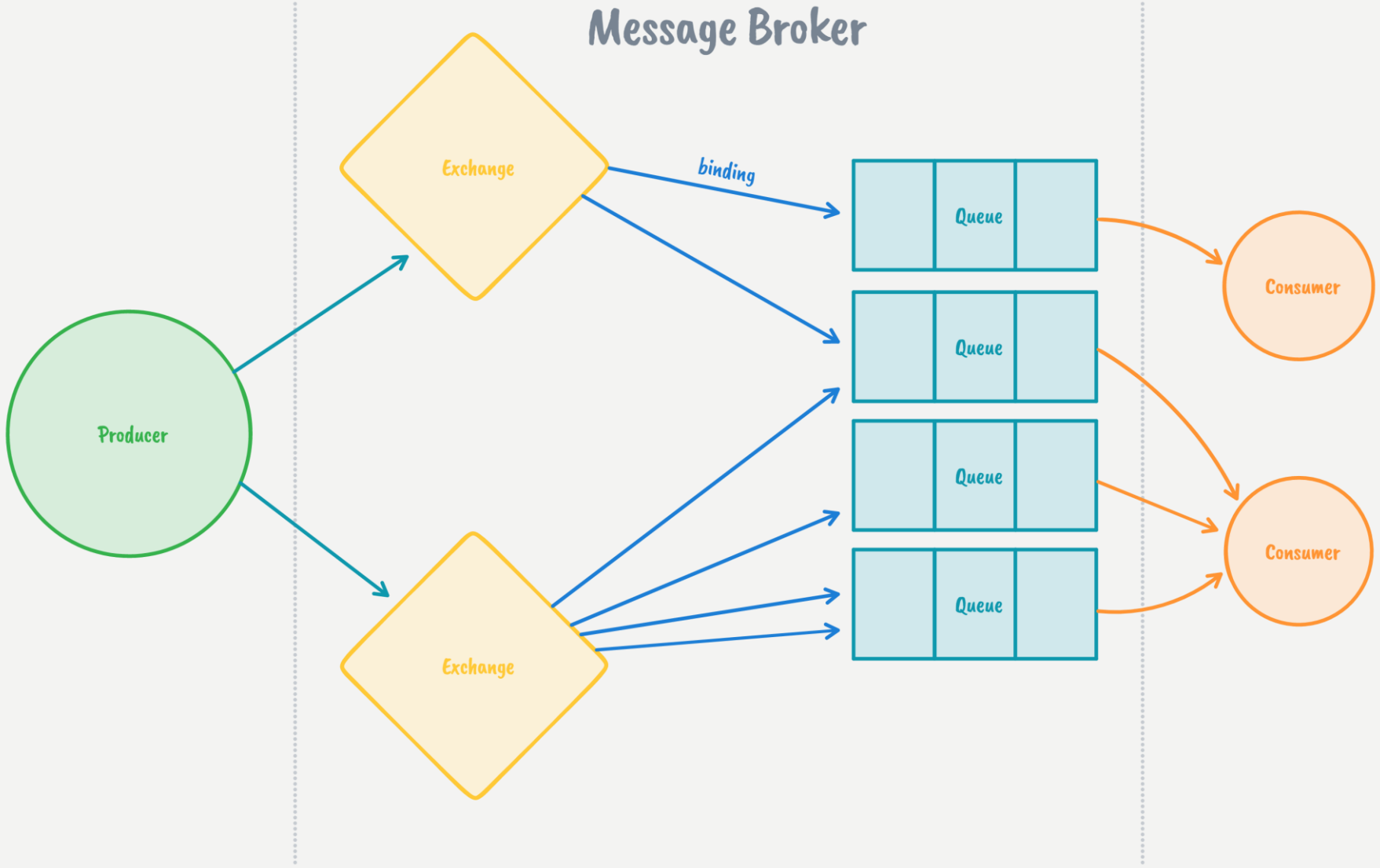
RABBITMQ

BRÈVE INTRODUCTION

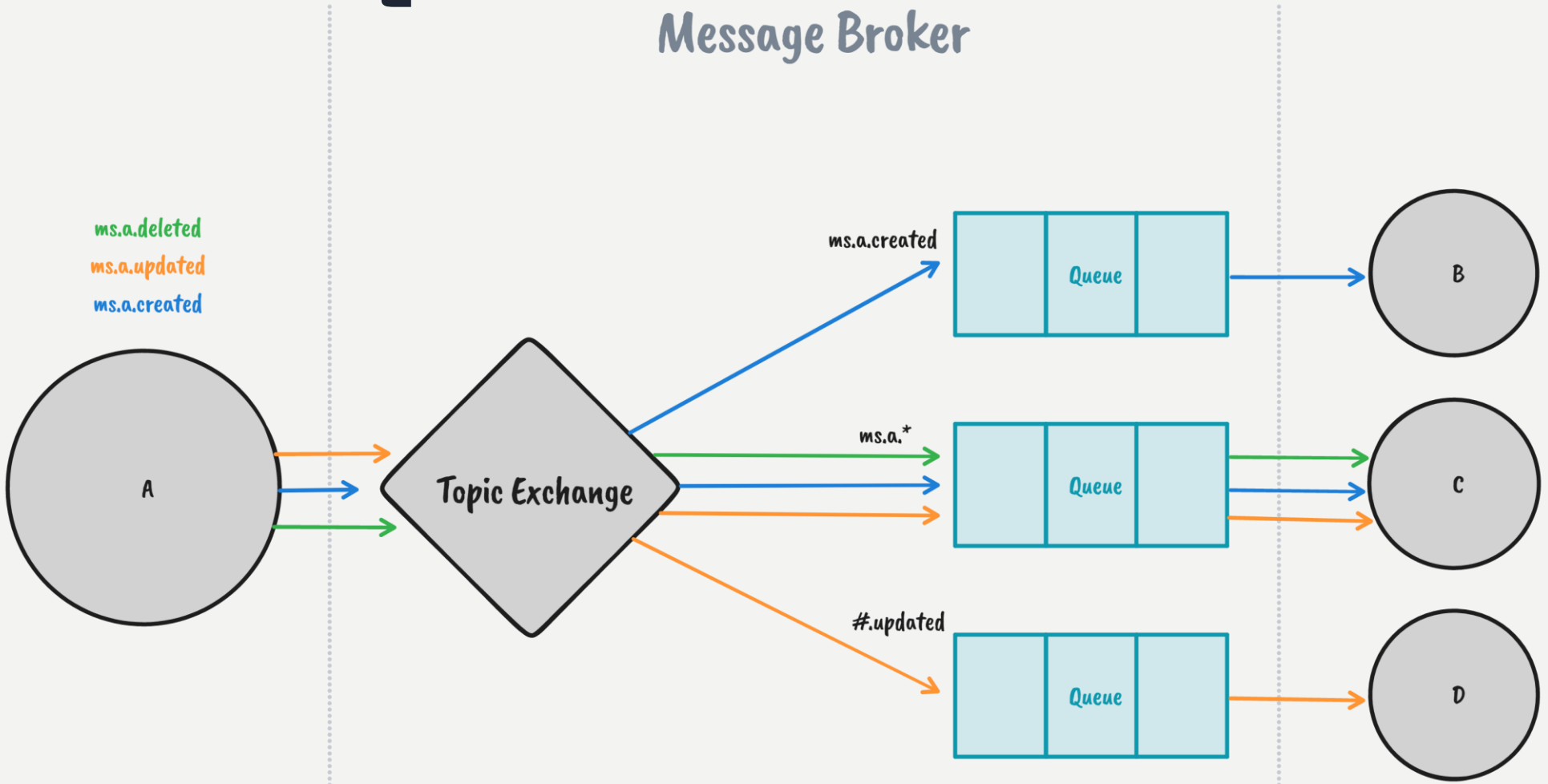
RABBITMQ

- RabbitMQ est un Message Broker
  - Gère des queues
    - Une queue est liée à un ou plusieurs exchanges
  - Gère des exchanges
    - Un exchange est lié à une ou plusieurs queues
  - Permet le rejeu
  - Permet la persistance d'une queue (si indisponible par exemple)

RABBITMQ


Message Broker
Exchange
binding
Queue
Queue
Queue
Queue
Consumer
Consumer
Producer
Exchange

RABBITMQ


Message Broker
ms.a.deleted
ms.a.updated
ms.a.created
A
Topic Exchange
ms.a.created
ms.a.*
#.updated
Queue
Queue
Queue
B
C
D

SPRING BOOT

EXEMPLE D'IMPLÉMENTATIONS

SPRING BOOT

- Ajouter la dépendance *spring-cloud-stream-binder-rabbit*

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

- Configurer la connexion dans le fichier *application.properties*

```
spring.rabbitmq.host = localhost
spring.rabbitmq.port = 5672
spring.rabbitmq.username = guest
spring.rabbitmq.password = guest
```

SPRING BOOT – PRODUCER

- Injecter **StreamBridge** à l'endroit nécessaire (dans un contrôleur par exemple)
- Utiliser le **StreamBridge** pour envoyer un évènement

```
this.streamBridge.send("nom-evenement-out-0", message);
```

- Configurer le topic de l'évènement dans le fichier *application.properties*

```
spring.cloud.stream.bindings.nom-evenement-out-0.destination = nom.exchange

spring.cloud.stream.rabbit.bindings.nom-evenement-out-0.producer.routing-key-expression = 'nom.cle'
```

SPRING BOOT – CONSUMER

- Déclarer un **Consumer**, ou un **Supplier**

```
@Bean
public Consumer<CustomTypeEvent> nomEvenement1() {
  return evt -> {
    System.out.println(evt);
  };
}
```

- Configurer les topics dans le fichier *application.properties*

```
spring.cloud.function.definition = nomEvenement1;nomEvenement2

spring.cloud.stream.bindings.nomEvenement1-in-0.destination = nom.exchange
spring.cloud.stream.bindings.nomEvenement1-in-0.group = nom.queue
spring.cloud.stream.rabbit.bindings.nomEvenement1-in-0.consumer.binding-routing-key = nom.cle
```

```
NOTE : Le nom de la queue complète sera destination.group
```

.NET6

EXEMPLE D'IMPLÉMENTATIONS

.NET6

- Ajouter les packages
  - *RabbitMQ.Client*
  - *Steeltoe.Connector.ConnectorCore*
  - *Steeltoe.Messaging.RabbitMQ*

```
<PackageReference Include="RabbitMQ.Client" Version="5.1.2" />
<PackageReference Include="Steeltoe.Connector.ConnectorCore" Version="3.2.1" />
<PackageReference Include="Steeltoe.Messaging.RabbitMQ" Version="3.2.1" />
```

.NET6

- Configurer le client RabbitMQ dans *appsettings.json*

```
"RabbitMq": {
  "Client": {
    "Uri": "amqp://guest:guest@localhost/"
  }
},
```

.NET6

- Ajouter le service dans *Program.cs*

```
builder.Services.AddRabbitMQConnection(builder.Configuration);
builder.Services.AddRabbitServices(true);
builder.Services.AddRabbitAdmin();
builder.Services.AddRabbitTemplate();
```

.NET6 – CONSUMER

- Ajouter le service dans *Program.cs*

```
builder.Services.AddSingleton<CustomEventHandler>();
builder.Services.AddRabbitListeners<CustomEventHandler>();
```

.NET6 – CONSUMER

- Déclarer les queues et bindings dans *Program.cs*
  - Ces déclarations peuvent être faites dans la classe Handler (voir plus loin)

```
builder.Services.AddRabbitQueue("nom.queue");

builder.Services.AddRabbitBinding("nomEvenement1", Binding.DestinationType.QUEUE, (p, b) => {
    var binding = b as QueueBinding;
    binding.Exchange = "nom.exchange";
    binding.Destination = "nom.queue";
    binding.RoutingKey = "nom.cle";
});
```

.NET6 – CONSUMER

- Créer **CustomEventHandler**

```
public class CustomEventHandler
{
  [RabbitListener(Binding = "nomEvenement1")]
  public void on(CustomTypeEvent e)
  {
      System.Console.WriteLine(e);
  }
}
```

.NET6 – CONSUMER

- Créer **CustomEventHandler** (et déclarer les queues et bindings)

```
public class CustomEventHandler
{
  [DeclareQueue(Name = "ms.nom.evenement.nomService", Durable = "True")]
  [DeclareQueueBinding(
    Name = "ms.nom.evenement.binding",
    QueueName = "ms.nom.evenement.nomService",
    ExchangeName = "ms.nom.evenement",
    RoutingKey = "#")
  ]
  [RabbitListener(Binding = "ms.nom.evenement.binding")]
  public void on(CustomTypeEvent e)
  {
      System.Console.WriteLine(e);
  }
}
```

.NET6 – PRODUCER

- Déclarer les exchanges dans *Program.cs*
  - Ces déclarations peuvent être faites par attribut

```
builder.Services.AddRabbitExchange("ms.commentaire", ExchangeType.TOPIC);
```

.NET6 – PRODUCER

- Injecter **RabbitTemplate** dans la classe qui aura besoin d'envoyer un message
- Utiliser **RabbitTemplate** pour envoyer un message

```
_rabbitTemplate.ConvertAndSend("nom.exchange", "nom.cle", message);
```