### 4.6 Variantes du patron Publication/Abonnement

Le patron de conception Publication/Abonnement (Pub/Sub) constitue une fondation architecturale puissante pour le découplage des composants dans les systèmes distribués. Il établit un contrat de communication asynchrone où les éditeurs (publishers) émettent des messages sans connaissance directe des abonnés (subscribers) qui les consommeront. Cependant, les exigences complexes des systèmes d'entreprise modernes — telles que l'intégrité transactionnelle, la résilience face aux pannes, le traitement de données à grande échelle et l'intégration inter-environnements — dépassent souvent les capacités du patron de base. Pour répondre à ces défis, plusieurs variantes et patrons complémentaires ont émergé. Ces spécialisations ne sont pas des alternatives, mais plutôt des extensions et des adaptations qui raffinent le modèle Pub/Sub pour des contextes spécifiques. Cette section présente une analyse détaillée de neuf de ces variantes fondamentales, en explorant leur finalité, leurs mécanismes internes et leur impact sur l'architecture globale d'un système.

#### 4.6.1 Courtier de messages centralisé (Centralized Message Broker)

**a. Explication et finalité**

Le **courtier de messages centralisé** est un composant logiciel intermédiaire qui agit comme un pivot pour la communication entre des applications et des services hétérogènes. Il s'agit de la matérialisation la plus courante du patron Pub/Sub dans une architecture logicielle.1 Sa finalité principale est d'instaurer un **découplage** fort entre les producteurs de messages (éditeurs) et leurs consommateurs (abonnés). Les éditeurs envoient des messages au courtier sans avoir besoin de connaître l'emplacement, le statut (en ligne ou hors ligne) ou le nombre d'abonnés.1 Réciproquement, les abonnés reçoivent des messages du courtier sans connaître l'identité de l'éditeur.

Ce découplage permet une communication **asynchrone**, où les systèmes peuvent continuer de fonctionner même en cas de latence réseau ou d'indisponibilité temporaire d'un des participants.3 Le courtier assume la responsabilité de la gestion du cycle de vie des messages, incluant leur validation, leur stockage temporaire, leur routage vers les destinations appropriées et leur livraison garantie.1 En centralisant ces fonctions de communication, le courtier simplifie le développement des applications individuelles, qui peuvent se concentrer sur leur logique métier plutôt que sur les complexités de la communication distribuée.

**b. Vulgarisation et analogie**

On peut comparer un courtier de messages centralisé à un **bureau de poste** moderne et hautement efficace.5 Lorsqu'un expéditeur (l'éditeur) veut envoyer une lettre (un message), il la dépose simplement dans une boîte aux lettres. L'expéditeur n'a pas besoin de savoir où se trouve le destinataire, quel est son horaire, ni quel facteur effectuera la livraison. Le bureau de poste (le courtier) prend en charge la totalité du processus : il collecte la lettre, la trie en fonction de l'adresse (le sujet ou la file d'attente), la transporte de manière fiable et la dépose dans la boîte aux lettres du destinataire (la file de l'abonné). Le destinataire peut alors récupérer son courrier à sa convenance, que l'expéditeur soit encore présent ou non. Cette analogie illustre parfaitement les concepts de découplage (l'expéditeur et le destinataire ne se rencontrent jamais), de communication asynchrone (la livraison n'est pas instantanée) et de stockage fiable (le bureau de poste garantit que la lettre ne sera pas perdue en chemin).

**c. Détails techniques**

Un courtier de messages est une solution logicielle complexe, souvent qualifiée de **Message-Oriented Middleware (MOM)**, qui repose sur plusieurs composants et mécanismes clés.

* **Composants clés** :
  + **Files d'attente (Queues)** : Utilisées principalement pour la communication point à point. Un message envoyé à une file d'attente est stocké jusqu'à ce qu'un seul consommateur le récupère et l'acquitte. Une fois consommé, il est retiré de la file.1
  + **Sujets (Topics)** : Le pilier du modèle Pub/Sub. Un message publié sur un sujet est dupliqué et distribué à tous les abonnés actifs qui ont manifesté leur intérêt pour ce sujet. Chaque abonné reçoit sa propre copie du message.3
  + **Échanges (Exchanges)** : Dans des courtiers comme RabbitMQ, un échange est une entité de routage plus sophistiquée. Les éditeurs publient des messages vers un échange, qui les achemine ensuite vers une ou plusieurs files d'attente en fonction de règles de liaison (bindings) et du type d'échange (direct, topic, fanout, headers).7
  + **Gestionnaires de files (Queue Managers)** : Composants internes responsables de la gestion des interactions entre les différentes files d'attente et de l'état des messages.1
* **Mécanismes internes** :
  + **Persistance des messages** : Pour garantir la durabilité, les messages peuvent être écrits sur disque. Ainsi, en cas de redémarrage du courtier, les messages ne sont pas perdus.1
  + **Garanties de livraison** : Les courtiers offrent différents niveaux de garantie : **at-most-once** (au plus une fois, risque de perte), **at-least-once** (au moins une fois, risque de duplication) et **exactly-once** (exactement une fois, le plus complexe à atteindre).
  + **Transactions** : Certains courtiers supportent des transactions qui permettent de regrouper l'envoi ou la réception de plusieurs messages en une seule opération atomique.
* **Défis techniques** :
  + **Point de défaillance unique (Single Point of Failure)** : De par sa nature centralisée, le courtier peut devenir un goulot d'étranglement ou un point de défaillance critique. Pour pallier ce risque, les déploiements en production utilisent des configurations en cluster à haute disponibilité.8
  + **Scalabilité** : Bien que les courtiers modernes soient conçus pour être hautement scalables, ils peuvent atteindre des limites de performance sous des charges extrêmes, comparativement à des architectures plus décentralisées comme les plateformes de streaming d'événements.1
  + **Complexité opérationnelle** : Le déploiement, la configuration, la surveillance et la maintenance d'un cluster de courtiers de messages exigent une expertise spécialisée pour assurer la performance et la fiabilité.

Un choix architectural fondamental lors de la sélection d'un courtier est de déterminer où la logique de routage doit résider. Cette décision mène à deux philosophies distinctes. D'une part, les **courtiers intelligents (smart brokers)** comme RabbitMQ centralisent une logique de routage complexe via des échanges sophistiqués, simplifiant ainsi les clients qui deviennent des **points d'extrémité simples (dumb endpoints)**.10 D'autre part, les **courtiers simples (dumb brokers)** comme Apache Kafka agissent principalement comme des journaux de transactions (logs) distribués et immuables, déléguant la logique de filtrage et de traitement aux **consommateurs intelligents (smart consumers)**.8 Ce choix a des implications profondes : un courtier intelligent peut accélérer le développement initial mais risque de concentrer la complexité dans l'infrastructure, tandis qu'un courtier simple favorise l'autonomie des services mais exige des implémentations clientes plus complexes.

**d. Implémentation dans le contexte Pub/Sub**

Le courtier de messages centralisé est l'implémentation la plus directe et la plus concrète du patron Pub/Sub. Il fournit l'infrastructure physique et logicielle qui matérialise le "canal de messages" abstrait du patron de base. La relation entre les composants est formalisée par le courtier :

* L'**éditeur** se connecte au courtier et publie un message sur un **sujet** nommé.
* L'**abonné** se connecte au courtier et crée une **souscription** à ce même sujet.
* Le **courtier** est responsable de copier chaque message arrivant sur le sujet et de le livrer à chaque souscription active, où il attend d'être consommé.6

Le courtier agit donc comme le tiers de confiance qui garantit le respect du contrat Pub/Sub, en isolant complètement les éditeurs des abonnés.

#### 4.6.2 Event Sourcing avec Pub/Sub

**a. Explication et finalité**

L'**Event Sourcing** (ou approvisionnement par les événements) est un patron de persistance des données qui se distingue radicalement des approches traditionnelles basées sur l'état (CRUD). Au lieu de stocker uniquement l'état actuel d'une entité, l'Event Sourcing enregistre chaque changement de cet état sous la forme d'une séquence d'**événements immuables**.11 L'état actuel d'une entité est donc une conséquence dérivée; il peut être reconstruit à tout moment en rejouant la séquence complète des événements qui la concernent depuis sa création.

La finalité principale de ce patron est de préserver une **piste d'audit complète et inaltérable** de toutes les actions survenues dans le système. Chaque événement est un fait métier qui ne peut être ni modifié ni supprimé. Cette approche offre une traçabilité parfaite, ce qui est essentiel dans des domaines réglementés comme la finance ou la santé. De plus, elle permet des fonctionnalités avancées telles que l'analyse temporelle ("quel était l'état de cette commande il y a trois jours?"), le débogage de logiques métier complexes et la reconstruction d'états passés.11

**b. Vulgarisation et analogie**

Imaginons la gestion d'un compte bancaire. Un système traditionnel (CRUD) ne conserverait que le solde actuel, par exemple 120 $. Si une erreur est suspectée, il est difficile de savoir comment on est arrivé à ce montant. L'Event Sourcing, en revanche, fonctionne comme le **grand livre comptable** de la banque.11 Il n'enregistre pas le solde, mais chaque transaction individuelle :

+Dépôt initial de 100 $, +Dépôt de 50 $, -Retrait de 20 $, -Achat par carte de 10 $. Le solde actuel de 120 $ peut être calculé à tout moment en additionnant et soustrayant toutes les transactions. Plus important encore, le grand livre offre une vision historique complète et auditable qui est bien plus riche que le simple solde final.

**c. Détails techniques**

L'implémentation de l'Event Sourcing repose sur un ensemble de composants et de concepts spécifiques.

* **Composants clés** :
  + **Commandes (Commands)** : Représentent une intention de modifier l'état du système (ex: PlacerCommandeCommand). Elles peuvent être acceptées ou rejetées par la logique métier.
  + **Événements (Events)** : Représentent un fait immuable qui s'est produit dans le passé (ex: CommandePlacéeEvent). Ils sont le résultat d'une commande validée.
  + **Magasin d'événements (Event Store)** : Une base de données spécialisée, optimisée pour l'ajout séquentiel (append-only). C'est la source unique de vérité du système; elle stocke tous les événements de manière ordonnée et durable.11
  + **Agrégats (Aggregates)** : Des entités métier (ex: une Commande) dont l'état est maintenu en mémoire. Pour prendre une décision, un agrégat rejoue ses propres événements afin de reconstituer son état actuel avant de valider une nouvelle commande.
  + **Projections (Projections / Read Models)** : Des vues de données optimisées pour la lecture. Elles sont construites en consommant le flux d'événements et en le transformant en un modèle de données dénormalisé, adapté aux besoins spécifiques des requêtes de l'interface utilisateur ou des API.11
* **Défis techniques** :
  + **Cohérence à terme (Eventual Consistency)** : Les projections (modèles de lecture) sont mises à jour de manière asynchrone après la persistance de l'événement. Il existe donc une brève période durant laquelle les données lues peuvent ne pas refléter le tout dernier changement, un compromis qui doit être géré au niveau de l'application.6
  + **Gestion des schémas (Schema Management)** : Les événements étant immuables, la modification de leur structure au fil du temps (versionnement des événements) est un défi majeur qui nécessite des stratégies de migration claires (ex: transformation à la volée, événements de mise à niveau).
  + **Complexité de l'interrogation** : Interroger directement le magasin d'événements pour obtenir des informations complexes sur l'état est souvent inefficace. La création et la maintenance de projections sont donc quasi obligatoires pour toute application non triviale.

**d. Implémentation dans le contexte Pub/Sub**

L'Event Sourcing et le Pub/Sub ne sont pas des alternatives mais des patrons hautement synergiques. L'Event Sourcing est un modèle de persistance, tandis que Pub/Sub est le mécanisme de communication qui permet de diffuser les fruits de cette persistance.11

Dans cette architecture, le **magasin d'événements (Event Store)** devient l'**éditeur** principal du système. Dès qu'un événement est écrit avec succès dans le magasin, celui-ci (ou un service de relais qui le surveille) le publie sur un courtier de messages.

Divers services en aval agissent alors comme **abonnés** à ce flux d'événements :

* Les **Projectors** s'abonnent pour construire et mettre à jour les modèles de lecture (projections), formant ainsi le côté "Query" d'une architecture CQRS.
* D'autres **microservices** s'abonnent pour réagir à des événements métier significatifs, ce qui est le fondement de la chorégraphie de microservices.

L'adoption de l'Event Sourcing transforme fondamentalement le rôle du patron Pub/Sub. Le flux d'événements n'est plus une simple série de notifications éphémères; il devient la **source de vérité faisant autorité** pour toute communication inter-services et pour la propagation de l'état à travers l'écosystème. Un message Pub/Sub classique pourrait notifier que "l'utilisateur 123 a été mis à jour", obligeant le destinataire à interroger un autre service pour obtenir les détails.13 Avec l'Event Sourcing, l'événement publié contient le changement lui-même : AdresseUtilisateurModifiée {rue: "123 rue Principale"}.11 Les services abonnés peuvent alors construire leur propre état local avec une confiance absolue dans l'exactitude des données reçues. Par conséquent, les garanties de fiabilité et d'ordre du système Pub/Sub deviennent d'une importance capitale, car elles ont un impact direct sur la cohérence de tous les systèmes en aval.

#### 4.6.3 Microservices en chorégraphie (Choreographed Microservices)

**a. Explication et finalité**

La **chorégraphie de microservices** est un style architectural décentralisé pour la coordination de processus métier qui s'étendent sur plusieurs services.14 Contrairement à l'approche par orchestration, où un composant central (l'orchestrateur) dicte explicitement chaque étape du flux de travail, la chorégraphie permet aux services de collaborer de manière autonome. Chaque service publie des événements pour signaler l'accomplissement de sa tâche, et les autres services intéressés s'abonnent à ces événements pour déclencher leurs propres actions en réponse.15 Le flux de travail émerge de cette série de réactions en chaîne, sans qu'aucun service n'ait une connaissance globale de l'ensemble du processus.

La finalité de cette approche est d'atteindre un **découplage maximal** et une **autonomie de service** élevée. En éliminant le coordinateur central, on supprime un point de défaillance unique et un goulot d'étranglement potentiel. Les services peuvent être développés, déployés et mis à l'échelle indépendamment les uns des autres, ce qui favorise l'agilité et la résilience du système global.14

**b. Vulgarisation et analogie**

On peut comparer la chorégraphie à une **troupe de danseurs de ballet** sur scène.14 Il n'y a pas de metteur en scène qui leur crie des instructions. Chaque danseur connaît sa partition et réagit à la musique (les événements) ainsi qu'aux mouvements de ses partenaires. Lorsqu'une danseuse termine une pirouette (un événement), le danseur à côté sait que c'est le signal pour commencer son saut. La performance globale est le résultat de ces interactions locales et autonomes. L'orchestration, en revanche, serait l'équivalent d'un réalisateur de film qui dirige chaque acteur individuellement, leur disant précisément quand et comment agir.

**c. Détails techniques**

L'architecture chorégraphiée repose sur une communication asynchrone et événementielle.

* **Composants clés** :
  + **Microservices** : Des services autonomes, chacun responsable d'une capacité métier distincte et possédant sa propre base de données.
  + **Courtier d'événements (Event Broker)** : Le système nerveux central de l'architecture. Il s'agit d'un courtier de messages (ex: RabbitMQ, Apache Kafka, Azure Service Bus) qui facilite la communication asynchrone via le patron Pub/Sub.14
* **Mécanismes internes** : Un processus métier se déroule comme suit :
  1. Le Service A exécute une opération (ex: enregistrer une nouvelle commande) et publie un événement CommandeCréée.
  2. Le Service B (Paiements) est abonné à CommandeCréée. À la réception de l'événement, il traite le paiement et publie un événement PaiementTraité.
  3. Le Service C (Expédition) est abonné à PaiementTraité. Il prépare le colis et publie CommandeExpédiée.  
     Le processus se termine lorsque plus aucun service ne réagit aux derniers événements publiés.
* **Défis techniques** :
  + **Visibilité du processus métier** : Sans un point de contrôle central, il devient difficile de suivre l'état d'un processus de bout en bout. Le débogage et la surveillance d'une transaction qui a échoué à mi-parcours peuvent être complexes, nécessitant des outils de traçabilité distribuée.15
  + **Gestion des transactions distribuées** : Assurer la cohérence des données à travers plusieurs services est le défi majeur. Si le Service B échoue après que le Service A a réussi, comment annuler l'opération du Service A? La solution implique souvent la mise en œuvre de patrons de compensation complexes comme les **Sagas**.15
  + **Dépendances cycliques** : Il existe un risque que les services créent des boucles de dépendance (A déclenche B, qui déclenche A), conduisant à des réactions en chaîne infinies. Une conception et une gouvernance rigoureuses des événements sont nécessaires pour éviter ce piège.

**d. Implémentation dans le contexte Pub/Sub**

La chorégraphie est l'application directe du patron Pub/Sub au niveau de l'architecture des systèmes. Le courtier d'événements et le modèle Pub/Sub ne sont pas de simples détails d'implémentation; ils sont le fondement même qui rend cette approche possible.

* Chaque microservice joue un double rôle : il est un **éditeur** lorsqu'il émet des événements pour communiquer les changements de son propre état, et un **abonné** lorsqu'il écoute les événements émis par d'autres services pour déclencher sa propre logique.
* Les **sujets** sur le courtier de messages deviennent le vocabulaire partagé du système. Ils représentent les faits métier significatifs (ex: ClientInscrit, ProduitAjoutéAuPanier, FactureGénérée) qui orchestrent implicitement le comportement global de l'application.

Le haut degré de découplage prôné par la chorégraphie est une arme à double tranchant. Il est la source de la flexibilité du système, mais il introduit directement le défi critique de la cohérence des données. Chaque service, pour participer à la chorégraphie, doit effectuer deux actions distinctes : mettre à jour sa propre base de données et publier un événement. Comme ces deux opérations ciblent des systèmes différents (une base de données transactionnelle et un courtier de messages), elles ne peuvent pas être englobées dans une transaction distribuée traditionnelle. C'est le **problème de la double écriture (dual-write problem)**.17 Une défaillance entre la validation de la transaction de la base de données et la publication de l'événement peut laisser le système dans un état incohérent, brisant silencieusement la chaîne chorégraphique. Par conséquent, une architecture en chorégraphie robuste ne peut être conçue sans une solution explicite à ce problème, rendant des patrons comme le Transactional Outbox (voir section 4.6.8) un compagnon quasi indispensable.

#### 4.6.4 Diffusion/Agrégation (Fan-Out / Fan-In)

**a. Explication et finalité**

Le patron **Diffusion/Agrégation** est une approche de traitement de données à grande échelle qui combine deux phases distinctes : **Fan-Out** (diffusion) et **Fan-In** (agrégation).

* **Fan-Out** : Cette phase consiste à décomposer une tâche volumineuse ou un message unique en de multiples sous-tâches ou messages plus petits, qui peuvent ensuite être traités de manière indépendante et en parallèle par plusieurs travailleurs.19
* **Fan-In** : Cette phase consiste à attendre que tous les travailleurs aient terminé leur traitement parallèle, puis à collecter (agréger) leurs résultats individuels pour produire un résultat final consolidé.19

La finalité de ce patron est d'améliorer drastiquement la performance et l'efficacité pour les tâches qui sont intrinsèquement divisibles. En tirant parti du parallélisme, il permet de réduire de manière significative le temps de traitement global pour des opérations comme le traitement par lots de données, les calculs complexes ou les appels à de multiples API externes.21

**b. Vulgarisation et analogie**

Imaginons l'organisation d'un grand banquet avec un menu complexe. Le chef cuisinier (le distributeur) reçoit la commande globale. Au lieu de tout préparer lui-même, il utilise la phase **Fan-Out** : il découpe la commande en tâches individuelles (préparer la salade, rôtir la viande, cuire les légumes) et les distribue à différents commis de cuisine (les travailleurs), qui s'exécutent tous en même temps. Une fois que chaque commis a terminé sa partie, la phase **Fan-In** commence : le chef attend que tous les plats soient prêts, puis il les rassemble, les dresse sur un plateau de service (agrégation) et envoie le repas complet au client. Cette approche parallèle est beaucoup plus rapide que si une seule personne avait dû tout faire séquentiellement. L'analogie du service de livraison de repas est également pertinente.21

**c. Détails techniques**

L'implémentation de ce patron nécessite une coordination minutieuse entre ses différents composants.

* **Composants clés** :
  + **Distributeur (Dispatcher / Producer)** : Le composant initial qui reçoit la tâche principale, la divise en sous-tâches et les publie pour traitement.
  + **Travailleurs (Workers / Consumers)** : Un groupe d'instances de service qui s'abonnent aux sous-tâches et les traitent en parallèle.
  + **Agrégateur (Aggregator)** : Le composant responsable de la phase Fan-In. Il doit détecter la fin du traitement de toutes les sous-tâches et combiner leurs résultats.
  + **Stockage partagé (Shared Storage)** : Un composant essentiel pour la coordination de la phase Fan-In. Comme les travailleurs sont indépendants, l'agrégateur a besoin d'un emplacement centralisé (ex: une base de données, un cache Redis, un compartiment de stockage d'objets comme S3) pour suivre l'état d'avancement de chaque sous-tâche et stocker les résultats partiels en attendant la consolidation finale.19
* **Mécanismes internes** : Le flux typique est le suivant :
  1. Le distributeur publie N messages, un pour chaque sous-tâche, souvent avec un identifiant de corrélation commun.
  2. N instances de travailleurs (ou moins, si la capacité est limitée) consomment ces messages et commencent le traitement.
  3. À la fin de son travail, chaque travailleur écrit son résultat dans le stockage partagé et signale son achèvement (ex: en incrémentant un compteur atomique dans Redis).
  4. L'agrégateur surveille le stockage partagé. Lorsqu'il détecte que les N sous-tâches sont terminées, il lit tous les résultats partiels, les combine pour former le résultat final, et potentiellement publie un événement de fin de processus.
* **Défis techniques** :
  + **Gestion de l'état** : La phase Fan-In est intrinsèquement avec état (stateful). La conception d'un mécanisme fiable pour suivre l'avancement et agréger les résultats ajoute une complexité significative par rapport à un traitement sans état.19
  + **Gestion des erreurs** : Que se passe-t-il si un travailleur échoue ou prend trop de temps? Le processus global doit-il échouer? Peut-il se terminer avec un résultat partiel? Des stratégies de nouvelles tentatives, de délais d'attente (timeouts) et de compensation doivent être mises en place.
  + **Fiabilité de la détection de fin** : L'agrégateur doit déterminer avec certitude que *toutes* les sous-tâches sont terminées. Dans un système distribué, cela peut être difficile à garantir sans risque de conditions de concurrence (race conditions).

**d. Implémentation dans le contexte Pub/Sub**

Le patron Diffusion/Agrégation utilise Pub/Sub comme une fondation, mais l'étend de manière significative.

* La phase **Fan-Out** est une application naturelle du patron Pub/Sub. Le distributeur publie des messages sur un sujet, et un groupe de consommateurs concurrents (competing consumers) s'abonnent à ce sujet pour se répartir la charge de travail.
* La phase **Fan-In**, cependant, va au-delà du modèle Pub/Sub de base. Le Pub/Sub est généralement sans état et unidirectionnel. Le Fan-In nécessite un chemin de retour pour les résultats et un mécanisme de gestion d'état pour la synchronisation. Cette logique doit être construite par-dessus l'infrastructure de messagerie, souvent en utilisant une file d'attente de réponse distincte où les travailleurs publient leurs résultats pour que l'agrégateur les consomme.

Ce patron démontre que Pub/Sub peut servir de brique de base pour construire des flux de travail complexes et avec état, qui peuvent même apparaître comme synchrones du point de vue de l'appelant initial. Il agit comme un pont entre le monde asynchrone de la messagerie et les exigences de synchronisation de nombreux processus métier. Les architectes peuvent ainsi utiliser ce patron pour résoudre des problèmes de performance sans abandonner une architecture événementielle, mais ils doivent être conscients que la complexité de la phase Fan-In (suivi d'état, gestion des erreurs, synchronisation) représente un effort de conception et d'implémentation non négligeable.

#### 4.6.5 Routage par sujet (Topic-Based Routing)

**a. Explication et finalité**

Le **routage par sujet** est une variante sophistiquée du patron Pub/Sub où les messages sont publiés sur des canaux nommés, les **sujets (topics)**, qui sont organisés selon une structure hiérarchique ou basée sur des patrons.7 Les consommateurs peuvent alors s'abonner non seulement à des sujets spécifiques, mais aussi à des ensembles de sujets en utilisant des **caractères génériques (wildcards)**. Par exemple, un consommateur pourrait s'abonner à ventes.electronique.\* pour recevoir toutes les ventes de produits électroniques, quel que soit le sous-type, ou à ventes.# pour recevoir toutes les ventes sans exception.23

La finalité de ce patron est d'offrir un mécanisme de **filtrage flexible et fin** côté infrastructure. Il permet aux abonnés de déclarer précisément le sous-ensemble de messages qui les intéresse, déchargeant ainsi les applications consommatrices de la tâche de recevoir et de filtrer un volume potentiellement énorme de messages non pertinents. Cela optimise l'utilisation des ressources réseau et de calcul.

**b. Vulgarisation et analogie**

Le routage par sujet peut être comparé à un **abonnement à un service de nouvelles personnalisé**. Une agence de presse (l'éditeur) publie des milliers d'articles chaque jour, en les étiquetant méticuleusement avec des catégories hiérarchiques, comme sport.hockey.canadiens ou politique.federal.elections. En tant que lecteur (l'abonné), vous n'êtes pas obligé de recevoir le journal entier. Vous pouvez configurer votre abonnement pour ne recevoir que les nouvelles qui vous intéressent : toutes les nouvelles sur le sport (sport.#), uniquement celles sur le hockey (sport.hockey.\*), ou même seulement les nouvelles concernant votre équipe favorite (sport.hockey.canadiens). Le kiosque à journaux (le courtier) se charge de vous livrer uniquement les sections que vous avez demandées.

**c. Détails techniques**

L'efficacité du routage par sujet dépend de la sémantique de la structure des sujets et des capacités du courtier.

* **Composants clés** :
  + **Sujets hiérarchiques** : Des noms de sujets structurés avec un délimiteur (généralement le point .) pour créer un espace de noms logique (ex: logs.production.service-paiement.erreur).24
  + **Clé de routage (Routing Key)** : Le nom de sujet exact avec lequel un message est publié (ex: logs.production.service-paiement.erreur).24
  + **Clé de liaison (Binding Key)** : Le patron, potentiellement avec des caractères génériques, utilisé par un abonné pour déclarer son intérêt (ex: logs.production.\*.erreur).24
  + **Caractères génériques (Wildcards)** :
    - L'astérisque (\*) correspond à exactement un mot (ex: a.\*.c correspond à a.b.c mais pas à a.b.d.c).
    - Le dièse (#) correspond à zéro ou plusieurs mots (ex: a.# correspond à a.b.c et à a).24
* **Mécanismes internes** : L'implémentation varie considérablement entre les différentes philosophies de courtiers.
  + **Courtiers "intelligents" (ex: RabbitMQ)** : Le routage est géré côté serveur. Un topic exchange reçoit les messages et effectue une correspondance de patrons entre la clé de routage du message et les clés de liaison des files d'attente qui lui sont associées. Le courtier est responsable de l'acheminement intelligent des messages.24
  + **Courtiers "simples" (ex: Apache Kafka)** : Le cœur de Kafka ne supporte pas nativement les abonnements par caractères génériques. Un message est publié sur un sujet unique et non divisible. Le filtrage est généralement une responsabilité du client : le consommateur s'abonne à un sujet plus large (ex: logs) et filtre les messages en mémoire. Une alternative consiste à utiliser un outil de traitement de flux (ex: Kafka Streams, ksqlDB) pour créer un nouveau sujet matériellement filtré en amont.27
* **Défis techniques** :
  + **Gouvernance des sujets** : L'efficacité du système repose entièrement sur une convention de nommage des sujets bien définie, cohérente et respectée par tous les développeurs de l'organisation. Une mauvaise gouvernance mène au chaos.
  + **Performance** : Dans les courtiers intelligents, une utilisation excessive de patrons de liaison complexes peut augmenter la charge de travail du courtier. Dans les courtiers simples, le filtrage côté client peut gaspiller de la bande passante et des cycles CPU si une grande partie des messages est rejetée.

**d. Implémentation dans le contexte Pub/Sub**

Le routage par sujet est une **spécialisation** puissante du patron Pub/Sub. Alors que le patron de base se contente de canaux nommés, cette variante enrichit le concept en introduisant une grammaire structurée et hiérarchique pour les noms de canaux, ainsi qu'un mécanisme de correspondance de patrons pour les abonnements. Il transforme une relation de publication simple en un système de filtrage de contenu sophistiqué, permettant des relations beaucoup plus granulaires et dynamiques entre les éditeurs et les abonnés.

Cette variante est un excellent exemple de la divergence philosophique entre les courtiers. Le choix de la technologie a un impact direct sur l'endroit où la logique de filtrage est implémentée. Avec un courtier comme RabbitMQ, l'intelligence est dans l'infrastructure; un service peut s'abonner à ventes.ca.\* et faire confiance au courtier pour ne lui livrer que les messages pertinents.24 Avec Kafka, le service doit s'abonner au sujet plus large ventes et implémenter lui-même la logique pour ne traiter que les messages concernant le Canada, ou alors dépendre d'une tâche de traitement de flux intermédiaire pour créer un sujet ventes\_ca pré-filtré.28 Cette décision n'est pas un simple détail d'implémentation, mais un compromis architectural fondamental entre la simplicité du consommateur et la complexité de l'infrastructure.

#### 4.6.6 Abonnés durables (Durable Subscribers)

**a. Explication et finalité**

Un **abonné durable** représente une souscription à un sujet qui persiste même lorsque l'application cliente qui la consomme est déconnectée.30 Dans un modèle de souscription non durable (ou transitoire), les messages publiés sur un sujet ne sont livrés qu'aux abonnés qui sont activement connectés au moment de la publication. Si un abonné est hors ligne, il manque tous les messages émis pendant son absence. L'abonné durable résout ce problème. Le courtier de messages est responsable de collecter et de stocker tous les messages destinés à une souscription durable, même en l'absence du client.

La finalité de ce patron est de **garantir la livraison des messages** à des abonnés qui peuvent avoir une connectivité intermittente ou des cycles de vie non continus (ex: applications mobiles, systèmes batch, services subissant des maintenances planifiées). Il assure qu'aucune donnée n'est perdue pendant les périodes de déconnexion, ce qui est crucial pour la fiabilité des processus métier asynchrones.30

**b. Vulgarisation et analogie**

On peut comparer une souscription durable à la location d'une **case postale**.30 Une souscription normale est comme la livraison du courrier à domicile : si vous n'êtes pas là pour le recevoir, il pourrait être laissé sur le perron et se perdre. Une souscription durable, c'est comme avoir une case postale sécurisée. Le bureau de poste (le courtier) y déposera tout votre courrier, que vous soyez en ville ou en vacances. Lorsque vous décidez de vous "connecter" (en vous rendant au bureau de poste), vous pouvez ouvrir votre case et récupérer en toute sécurité tout le courrier qui s'est accumulé pendant votre absence.

**c. Détails techniques**

La mise en œuvre des abonnés durables impose des exigences spécifiques tant au client qu'au courtier.

* **Composants clés** :
  + **Identifiant de client unique (Unique Client ID)** : Pour que le courtier puisse associer une connexion à une souscription persistante, le client doit s'identifier de manière unique et stable à chaque connexion. Cet identifiant permet au courtier de reconnaître le client qui revient en ligne et de lui livrer les messages en attente.30
  + **Nom de souscription unique** : La souscription elle-même doit avoir un nom unique qui la distingue des autres souscriptions durables sur le même sujet.
  + **Stockage persistant côté courtier** : Le courtier doit disposer d'un mécanisme de stockage fiable (généralement sur disque) pour conserver les messages accumulés pour chaque souscription inactive. Cette persistance doit survivre aux redémarrages du courtier.31
  + **Politiques de rétention (Retention Policies)** : Les messages ne sont pas conservés indéfiniment. Le courtier ou le sujet est configuré avec une durée de rétention (ex: 7 jours). Si un abonné ne se reconnecte pas dans ce délai, les messages accumulés peuvent être supprimés pour éviter l'épuisement des ressources.32
* **Mécanismes internes** :
  1. Un client se connecte au courtier avec un identifiant unique et crée une souscription durable avec un nom spécifique sur un sujet donné.
  2. Le client peut se déconnecter.
  3. Le courtier continue de recevoir les messages publiés sur le sujet et en stocke une copie pour cette souscription durable spécifique.
  4. Lorsque le client se reconnecte en utilisant le même identifiant et en faisant référence à la même souscription, le courtier commence à lui livrer tous les messages qui ont été stockés, en respectant l'ordre de publication.30
  5. Une fois les messages acquittés par le client, le courtier les supprime de son stockage.
* **Défis techniques** :
  + **Épuisement des ressources** : Le défi le plus important est la gestion du cycle de vie des souscriptions. Une souscription "oubliée" qui n'est jamais supprimée (unsubscribed) continuera d'accumuler des messages, consommant de l'espace disque et de la mémoire sur le courtier. Une gouvernance stricte est nécessaire pour nettoyer les souscriptions obsolètes.30
  + **Pic de charge au redémarrage (Catch-up Burst)** : Un abonné qui a été déconnecté pendant une longue période peut recevoir une avalanche de messages à sa reconnexion. L'application doit être conçue pour absorber ce pic de charge sans être submergée.

**d. Implémentation dans le contexte Pub/Sub**

Cette variante modifie une hypothèse fondamentale du patron Pub/Sub de base, à savoir que les abonnés doivent être actifs pour recevoir des messages. Elle introduit le concept de **souscriptions avec état (stateful subscriptions)**, où l'état de la souscription (y compris les messages non lus) est géré et maintenu par le courtier, indépendamment de l'état du client.

Dans un modèle Pub/Sub non durable, la responsabilité du courtier se limite à livrer les messages aux abonnés actuellement connectés. Avec le modèle durable, cette responsabilité s'étend pour inclure le stockage à long terme et la gestion d'état pour le compte de ses clients. Cela change fondamentalement le contrat entre l'abonné et le courtier, exigeant une identification unique et une gestion explicite du cycle de vie (création et suppression de la souscription).

L'utilisation de la durabilité transforme le courtier de messages d'un simple routeur de messages sans état en un magasin de données temporaire et avec état pour ses clients. Cela introduit un compromis direct : en échange d'une garantie de livraison plus forte, on accepte une augmentation de la complexité opérationnelle et des coûts de ressources. Le choix d'utiliser des abonnés durables n'est donc pas une simple option technique, mais un engagement opérationnel qui nécessite une surveillance robuste des ressources du courtier et une gouvernance rigoureuse du cycle de vie des souscriptions.

#### 4.6.7 Maillage d'événements (Event Mesh)

**a. Explication et finalité**

Un **maillage d'événements (Event Mesh)** est une couche d'infrastructure d'application dynamique et configurable, conçue pour router des événements entre des applications et des services répartis sur des environnements hétérogènes et géographiquement distribués (ex: centres de données sur site, clouds privés, multiples clouds publics).33 Plutôt que de reposer sur un unique courtier centralisé, un maillage est constitué d'un **réseau de courtiers d'événements interconnectés** qui collaborent pour former une fabrique de distribution d'événements unifiée et transparente.35

La finalité d'un maillage d'événements est de permettre une communication événementielle en temps réel, fluide et à l'échelle de l'entreprise, en éliminant les silos technologiques et géographiques. Il permet à un événement produit dans un environnement (ex: une application microservices sur Kubernetes dans AWS) d'être consommé de manière native par une autre application dans un environnement complètement différent (ex: un système hérité dans un centre de données sur site), sans que les applications n'aient à se soucier de la topologie du réseau, des protocoles ou des frontières entre les clouds.33

**b. Vulgarisation et analogie**

On peut imaginer le maillage d'événements comme **l'Internet des événements d'entreprise**.35 Lorsque votre ordinateur envoie un paquet de données, il ne se soucie pas du chemin complexe que celui-ci empruntera à travers des dizaines de routeurs pour atteindre un serveur à l'autre bout du monde. De même, avec un maillage d'événements, une application dans un "quartier" de votre entreprise (ex: une région cloud) peut publier un événement. Le maillage, agissant comme un réseau intelligent de routeurs d'événements, trouvera dynamiquement le chemin le plus efficace pour livrer cet événement à une application consommatrice dans un autre "quartier" (ex: une usine connectée ou un autre fournisseur de cloud), de manière transparente pour l'expéditeur et le destinataire.

**c. Détails techniques**

Un maillage d'événements est une architecture de réseau avancée pour la messagerie.

* **Composants clés** :
  + **Courtiers d'événements (Event Brokers)** : Ce sont les nœuds du maillage, déployés dans chaque environnement significatif (cloud, sur site, en périphérie). Des produits comme Solace PubSub+, HiveMQ ou des fédérations de courtiers open-source peuvent servir de base.35
  + **Routage dynamique des messages (Dynamic Message Routing)** : C'est l'intelligence fondamentale du maillage. Les courtiers interconnectés partagent des informations sur les sujets et les souscriptions. Lorsqu'un abonné se connecte à un courtier, cette information est propagée à travers le maillage. Ainsi, les événements sont acheminés uniquement vers les courtiers qui ont des consommateurs intéressés, optimisant ainsi le trafic réseau.35
  + **Ponts multi-protocoles (Multi-Protocol Bridges)** : La capacité du maillage à accepter et à livrer des événements en utilisant une variété de protocoles standards (ex: AMQP, MQTT, REST/Webhooks) et à les traduire si nécessaire. Cela permet à des applications modernes et à des systèmes hérités de communiquer de manière transparente.34
* **Mécanismes internes** : Les courtiers sont connectés les uns aux autres pour former une topologie de réseau. Lorsqu'un consommateur dans l'environnement B s'abonne à un sujet, son courtier local propage cette information de souscription aux autres courtiers du maillage. Lorsqu'un producteur dans l'environnement A publie un message sur ce sujet, son courtier local sait (grâce à la souscription propagée) qu'il doit router le message à travers le maillage vers le courtier de l'environnement B, qui le livre ensuite au consommateur. Le maillage est **auto-apprenant** (il découvre les chemins de routage) et **auto-réparateur** (il peut contourner les pannes de liens ou de nœuds).35
* **Défis techniques** :
  + **Gouvernance et sécurité centralisées** : La gestion d'un réseau distribué de courtiers nécessite des outils centralisés pour la gouvernance des sujets, la gestion des politiques de sécurité, le contrôle d'accès et la découverte de schémas d'événements.
  + **Latence inter-environnements** : Le routage d'événements à travers des réseaux étendus (WAN) ou entre différents fournisseurs de cloud introduit une latence inévitable qui doit être prise en compte dans la conception des applications.
  + **Complexité de déploiement et coût** : La mise en place et la gestion d'un réseau multi-cloud de courtiers d'événements représentent un investissement infrastructurel et opérationnel significatif.

**d. Implémentation dans le contexte Pub/Sub**

Le maillage d'événements est l'évolution logique du patron du courtier centralisé pour répondre aux défis des entreprises distribuées, hybrides et multi-cloud. On peut le considérer comme le **"Pub/Sub à l'échelle mondiale"**. Il prend les concepts fondamentaux de Pub/Sub (éditeurs, abonnés, sujets) et distribue l'infrastructure du courtier elle-même en un réseau coopératif et décentralisé. Cela permet à un éditeur connecté à un nœud du maillage de publier un événement qui sera reçu de manière transparente par un abonné connecté à un autre nœud, potentiellement à des milliers de kilomètres de distance.

Il est important de noter que le maillage d'événements et le **maillage de services (Service Mesh)** sont des patrons complémentaires et non concurrents.34 Un maillage de services (ex: Istio) se concentre sur la gestion de la communication synchrone (requête/réponse, appels API) entre les microservices, en fournissant des fonctionnalités comme la découverte de services, l'équilibrage de charge et la résilience. Un maillage d'événements, quant à lui, gère la communication asynchrone (événementielle). Ensemble, ils forment une couche de communication complète pour une entreprise moderne, couvrant à la fois les interactions de type "demander et attendre" (Service Mesh) et celles de type "déclencher et oublier" (Event Mesh).

#### 4.6.8 Patron Outbox (Transactional Outbox Pattern)

**a. Explication et finalité**

Le patron **Transactional Outbox** est une solution technique conçue pour garantir la fiabilité de la communication dans les systèmes distribués en assurant l'**atomicité** entre une modification de la base de données et la publication d'un message ou d'un événement correspondant.17 Il s'attaque directement au **problème de la double écriture (dual-write problem)**, qui survient lorsqu'une opération métier nécessite à la fois une écriture dans une base de données et l'envoi d'un message à un courtier. Sans ce patron, une défaillance entre ces deux actions peut laisser le système dans un état incohérent (la donnée est sauvegardée mais l'événement n'est jamais envoyé, ou vice-versa).17

La finalité de ce patron est de garantir qu'un événement est publié **si et seulement si** la transaction de base de données qui le génère est validée avec succès. Il transforme le problème de la double écriture en un problème de livraison garantie, qui est plus simple à résoudre.18

**b. Vulgarisation et analogie**

Imaginez que vous devez accomplir deux tâches indissociables : signer un contrat important et l'envoyer par courrier recommandé. L'approche naïve serait de signer le contrat, puis de vous rendre au bureau de poste. Mais que se passe-t-il si vous avez un accident en chemin? Le contrat est signé, mais il ne sera jamais envoyé. Le patron Outbox propose une meilleure approche. Au moment de signer le contrat, vous le placez immédiatement dans une **"boîte d'envoi" (outbox)** spéciale et verrouillée sur votre bureau. Ces deux actions (signer et mettre dans la boîte) se font en une seule étape atomique, sous la supervision d'un notaire (la transaction de la base de données). Ensuite, un service de messagerie fiable et indépendant (le relais de messages) a pour unique tâche de vérifier périodiquement votre boîte d'envoi, de prendre les lettres qui s'y trouvent et de les poster. Si vous décidez d'annuler et de déchirer le contrat avant la fin, la lettre n'atteint jamais la boîte d'envoi. Les deux actions sont ainsi garanties d'être exécutées ensemble, ou pas du tout.

**c. Détails techniques**

Le patron Outbox s'implémente en intégrant la publication de messages dans le cycle de vie transactionnel de la base de données.

* **Composants clés** :
  + **Table Outbox** : Une table dédiée (outbox) au sein de la même base de données que les données métier. Cette table stocke les messages ou événements qui doivent être publiés. Chaque enregistrement contient le contenu du message, sa destination (sujet), et un statut (ex: 'non publié').17
  + **Relais de messages (Message Relay)** : Un processus ou un service indépendant qui surveille la table outbox. Sa responsabilité est de lire les messages non publiés, de les envoyer au courtier de messages, et, en cas de succès, de marquer les enregistrements correspondants comme 'publiés' (ou de les supprimer) dans la table outbox.17
* **Mécanismes internes** :
  1. L'application démarre une transaction de base de données.
  2. Elle effectue les modifications sur les tables métier (ex: insérer une ligne dans la table Commandes).
  3. Dans la **même transaction**, elle insère un enregistrement dans la table outbox décrivant l'événement à publier (ex: CommandeCrééeEvent).
  4. L'application valide (commit) la transaction. Grâce à l'atomicité de la base de données, soit les deux écritures (données métier et outbox) réussissent, soit les deux échouent.
  5. Indépendamment, le relais de messages interroge (poll) la table outbox pour de nouveaux enregistrements.
  6. Lorsqu'il en trouve, il les publie sur le courtier de messages.
  7. Après avoir reçu une confirmation du courtier, il met à jour ou supprime l'enregistrement de la table outbox pour éviter une republication.17 Une alternative à l'interrogation est d'utiliser la Capture de Données modifiées (Change Data Capture - CDC), où le relais écoute directement le journal des transactions de la base de données pour détecter les nouvelles insertions dans la table outbox.17
* **Défis techniques** :
  + **Duplication de messages** : Le principal compromis. Le relais pourrait tomber en panne juste après avoir publié un message mais avant de mettre à jour la table outbox. Au redémarrage, il publiera le même message une seconde fois. Ce comportement de "livraison au moins une fois" (at-least-once) impose que tous les consommateurs de ces messages soient **idempotents** (c'est-à-dire capables de traiter le même message plusieurs fois sans effets de bord indésirables).18
  + **Latence de publication** : L'approche par interrogation introduit un délai entre la validation de la transaction et la publication effective de l'événement.
  + **Contention sur la table Outbox** : Dans les systèmes à très haut débit, la table outbox peut devenir un point de contention en écriture.36

**d. Implémentation dans le contexte Pub/Sub**

Le patron Outbox est un compagnon essentiel du patron Pub/Sub dans tout système qui requiert une intégrité transactionnelle. Il ne remplace pas Pub/Sub, mais il **fiabilise l'étape de "publication"**.

Il modifie le flux Pub/Sub standard en introduisant une étape intermédiaire et asynchrone. Au lieu que la logique métier de l'application publie directement sur le courtier, elle "publie" dans une table de sa base de données locale. C'est ensuite le relais de messages qui devient le véritable **éditeur** pour le système Pub/Sub. Cette approche découple la transaction métier de l'interaction (potentiellement faillible) avec un service réseau externe comme le courtier, échangeant l'immédiateté contre la sécurité transactionnelle.

Ce patron a une conséquence architecturale en cascade. En résolvant le problème d'atomicité, il déplace le défi vers la garantie de livraison, qui est assurée par le relais avec une sémantique "au moins une fois". Cette garantie de livraison, à son tour, impose une exigence stricte d'idempotence à tous les systèmes en aval. Un architecte qui choisit le patron Outbox a donc la responsabilité de s'assurer que *chaque service* qui s'abonne aux événements publiés via ce mécanisme est conçu pour être idempotent. Cette exigence n'est pas une option, mais une conséquence non négociable de la recherche de fiabilité côté éditeur.

#### 4.6.9 CQRS avec Pub/Sub

**a. Explication et finalité**

**CQRS**, ou **Command Query Responsibility Segregation** (Ségrégation des responsabilités de commande et de requête), est un patron architectural qui prône la séparation des modèles utilisés pour modifier l'état d'un système (le côté **Commande**) de ceux utilisés pour lire cet état (le côté **Requête**).37 Dans une architecture traditionnelle (CRUD), un seul modèle de données est utilisé pour les opérations de création, lecture, mise à jour et suppression. Le CQRS rompt avec cette approche en utilisant des modèles distincts, optimisés pour chaque tâche.

* Le **côté Commande** (write side) est responsable du traitement des commandes, de l'application de la logique métier et de la validation des règles. Il est optimisé pour la cohérence transactionnelle.
* Le **côté Requête** (read side) est responsable de fournir des données aux clients. Il utilise des modèles de lecture (read models) souvent dénormalisés et optimisés pour des performances de lecture rapides.

La finalité du CQRS est d'optimiser et de faire évoluer chaque côté indépendamment. Le côté écriture peut être complexe, avec une logique métier riche, tandis que le côté lecture peut être simple, rapide et facilement mis à l'échelle en ajoutant des répliques.37

**b. Vulgarisation et analogie**

On peut comparer une architecture CQRS à un **restaurant bien organisé**. La **cuisine** est le **côté Commande**. C'est là que les commandes des clients (les commandes) sont reçues. Les chefs (la logique métier) suivent des recettes strictes pour préparer les plats (modifier l'état). La cuisine est la source unique de vérité pour la création des repas et est optimisée pour la cohérence et la qualité. Le **menu** présenté aux clients est le **côté Requête**. C'est une vue simplifiée, dénormalisée et facile à lire de ce qui est disponible, optimisée pour que les clients puissent faire leur choix rapidement. Lorsque le chef décide qu'un plat n'est plus disponible (un changement d'état du côté Commande), il envoie une notification (un événement via Pub/Sub) au personnel de salle pour qu'il mette à jour les menus. Les clients ne consultent jamais directement l'inventaire de la cuisine; ils lisent toujours le menu optimisé.

**c. Détails techniques**

Une architecture CQRS distribuée repose sur la communication asynchrone pour maintenir la synchronisation.

* **Composants clés** :
  + **Côté Commande (Command Side)** : Comprend des **Command Handlers** qui reçoivent les commandes, chargent les agrégats du domaine, exécutent la logique métier et, en cas de succès, génèrent des événements.37
  + **Côté Requête (Query Side)** : Comprend des **Query Handlers** qui interrogent des **modèles de lecture (Read Models)**. Ces modèles peuvent être stockés dans des technologies variées et optimisées pour la lecture (ex: une base de données SQL, un cache, un moteur de recherche comme Elasticsearch).37
  + **Bus d'événements (Event Bus)** : Un mécanisme Pub/Sub qui sert de pont entre le côté Commande et le côté Requête. C'est le canal par lequel les changements d'état sont communiqués.37
* **Mécanismes internes** :
  1. Une interface utilisateur ou un service externe envoie une commande (ex: ChangerAdresseClientCommand) au côté Commande.
  2. Le Command Handler approprié traite la commande, applique les règles métier sur le modèle de domaine (l'agrégat Client).
  3. Si la commande est valide, l'état de l'agrégat est modifié et un ou plusieurs événements sont produits (ex: AdresseClientChangéeEvent).
  4. Ces événements sont publiés sur le bus d'événements.
  5. Des **Event Handlers** du côté Requête, abonnés à ces événements, les reçoivent et mettent à jour les modèles de lecture pertinents (ex: la table VueClients et la table AdressesDeLivraison).37
* **Défis techniques** :
  + **Cohérence à terme (Eventual Consistency)** : C'est le principal compromis du CQRS. Comme la mise à jour des modèles de lecture est asynchrone, il y a un délai pendant lequel les données lues peuvent être obsolètes. L'interface utilisateur et les clients de l'API doivent être conçus pour gérer cette éventualité.6
  + **Complexité accrue** : Le CQRS introduit plus de concepts et de composants (commandes, requêtes, événements, handlers multiples, modèles de données dupliqués) qu'une architecture CRUD, ce qui peut augmenter la complexité du code et la courbe d'apprentissage.
  + **Fiabilité de la synchronisation** : Le mécanisme de propagation des événements doit être extrêmement fiable. Une défaillance dans la livraison des événements peut entraîner une désynchronisation permanente entre les modèles d'écriture et de lecture.

**d. Implémentation dans le contexte Pub/Sub**

Dans une architecture CQRS distribuée, le patron Pub/Sub n'est pas une option, il est le **mécanisme de synchronisation fondamental** qui relie les deux côtés. Sans lui, la ségrégation ne serait pas possible de manière découplée.

* Le **côté Commande** agit en tant qu'**éditeur**. Après avoir traité avec succès une commande et modifié son état, il publie un événement qui décrit ce qui s'est passé.
* Les **Event Handlers du côté Requête** sont les **abonnés**. Ils écoutent les événements pertinents et sont responsables de la mise à jour de leurs modèles de lecture spécialisés.

Le CQRS n'est souvent pas un choix architectural initial, mais plutôt une nécessité qui émerge pour résoudre les problèmes de performance ou de complexité d'un domaine. Par exemple, l'adoption de l'Event Sourcing (section 4.6.2) conduit presque naturellement au CQRS, car le magasin d'événements est optimisé pour l'écriture et difficile à interroger directement, ce qui rend la création de modèles de lecture séparés (projections) indispensable.11 Dans ces scénarios, Pub/Sub est la solution qui rend cette séparation viable. Il permet au côté écriture de diffuser les changements sans connaître le nombre ou le type de modèles de lecture qui en dépendent, offrant une grande extensibilité à l'architecture. Un nouvel écran dans l'application nécessitant une vue de données différente peut être implémenté en ajoutant simplement un nouveau modèle de lecture et un nouvel abonné, sans aucune modification du côté Commande.