Contrôle de processus

Romain De Groote

16 octobre 2026

Un petit programme, reconstruit pas à pas de l'appel de méthode direct à une architecture événementielle sur MQTT : ce qui bouge à chaque étape, et qui finit par connaître qui.

Table des matières

À propos de ces notes **5**
Comment lire les figures . . . 5

1 Étape 0 — De quoi il s'agit **6**
1.1 Les trois questions . . . 6

2 Partie I — Commencer par ce que l'on connaît déjà **7**
2.1 Étape 1 — Au plus simple : appeler, point . . . 7
2.2 Étape 2 — Trois capacités égoïstes . . . 8
2.3 Étape 3 — Donner une boîte aux lettres à chaque capacité . . . 8
2.4 Étapes 4–6 — Les trois capacités deviennent égoïstes . . . 8
2.5 Étape 7 — Entre le médiateur . . . 9
2.6 Étape 8 — Redemander : qui connaît qui ? . . . 10
2.7 Étape 9 — Pourquoi des threads virtuels ? . . . 11
2.8 Étape 10 — Garder les services égoïstes . . . 11
2.9 Ce que la partie I a déplacé . . . 11

3 Partie II — Franchir la frontière de processus **13**
3.1 Étape 11 — Séparer les composants . . . 13
3.2 Étapes 12–13 — Un répertoire devient une boîte aux lettres . . . 13
3.3 Étape 14 — Le premier problème de système distribué : la publication . . . 14
3.4 Étape 15 — Observer une boîte avec `WatchService` . . . 14
3.5 Étapes 16–17 — Services et médiateur deviennent des processus . . . 14
3.6 Étape 18 — Exécuter la version entièrement externe, en local . . . 15
3.7 Ce que la partie II a déplacé . . . 16

4 Partie III — Le rendre réellement distribué **18**
4.1 Syncthing réplique ; il ne distribue pas . . . 18
4.2 Tester une couche à la fois . . . 18
4.3 Ce que coûte une frappe . . . 18
4.4 Casser un médiateur . . . 19
4.5 Le médiateur devient un service ordinaire . . . 19
4.6 Ce que la partie III a déplacé . . . 20

5 Partie IV — Rembobinage **21**
5.1 Le raccourci évident, et son piège . . . 21
5.2 La capacité n'est pas la topologie . . . 21
5.3 Ce qui change dans le code . . . 21
5.4 `ROUTE` et `TOPIC` sont de l'Intent . . . 22

6 Partie V — Rejouer l'idée dynamique entre processus **23**
6.1 Une observation bloquante ne suffit pas . . . 23
6.2 L'utilitaire : `FileSubscriptions` . . . 23
6.3 Deux publications au lieu de quatre . . . 24

7 Partie VI — Rejouer sur Syncthing, et nommer les messages **25**
7.1 L'exécution dynamique distribuée . . . 25
7.2 Ce que « direct » veut dire, exactement . . . 25
7.3 Intent, Status, Event . . . 26
7.4 La règle de l'écrivain unique . . . 26

8 Partie VII — Fusionner les deux idées architecturales **27**
8.1 Un transformateur mérite d'exister . . . 27
8.2 Le transformateur est égoïste aussi . . . 27
8.3 L'architecture complète sur système de fichiers . . . 28

9 Partie IX — MQTT **29**
9.1 Ce que le système de fichiers a enseigné, et ce pour quoi il n'est pas fait . . . 29
9.2 MQTT devrait sembler familier . . . 29
9.3 La bibliothèque, l'identifiant client et l'adresse du courtier . . . 30
9.4 Intent structuré, et deux mappeurs . . . 30
9.5 DUISE, et recâbler un système en marche . . . 31
9.6 Ce qu'un nouvel abonné apprend, et ce qu'un mort avoue . . . 31
9.7 Ce qui a survécu au transport, et le couplage restant . . . 32
9.8 Le parcours complet . . . 32
9.9 Les phrases à retenir . . . 33

Annexes **34**

Glossaire **34**

À propos de ces notes

Ces notes suivent un petit programme à travers une longue série de reconstructions, de l'appel de méthode direct à une architecture événementielle sur MQTT. Chaque reconstruction laisse le programme faire exactement ce qu'il faisait avant, et change qui connaît qui.

Le programme construit ne change jamais : vous tapez une ligne, elle revient décalée de 13 lettres.

`keyboard → ROT13 → console`

Ce qui change, c'est tout ce qui l'entoure. Le même système est reconstruit encore et encore, et chaque reconstruction pose trois questions, dans cet ordre :

Qui connaît qui ?

Qui possède quoi ?

Qui décide comment le système est câblé ?

Un résumé est ajouté ici chaque fois qu'une étape est terminée : le livre grandit donc au rythme du travail. Chaque résumé dit ce que l'étape a fait, ce qui a été observé à l'exécution, et ce qui a bougé sur le plan architectural — en général une connaissance qui a changé de mains.

D'après le matériel de Reto Konig, Université de Fribourg.

Comment lire les figures

Chaque figure existe une fois comme dessin et trois fois comme langue. Dans la version web, une figure qui porte un flux de messages possède une petite barre de lecture : **Lancer** fait parcourir toute la chaîne à un message, **Un saut** avance d'une seule étape, et **Réinitialiser** remet tout à zéro. Les boîtes s'allument au passage du message : on voit ainsi quel composant attend et lequel travaille.

Dans le PDF, et dans un navigateur sans JavaScript, la même image est là — simplement immobile. Rien dans ces notes ne dépend de l'animation.

1 Étape 0 — De quoi il s'agit

L'étape 0 ne contient aucun code. Elle énonce le sujet : non pas ROT13, mais l'architecture autour de lui. La fonctionnalité reste presque inchangée pendant 81 étapes, et c'est précisément ce qui rend visibles les changements alentour.

Deux règles sont posées ici et valent jusqu'au bout :

Ne rencontrer une solution qu'après avoir vécu le problème qu'elle résout.

C'est pourquoi la construction ne commence ni par MQTT, ni par les abonnements dynamiques. Un système aux médiateurs délibérément maladroits est bâti d'abord, mis en service, et l'on demande seulement ensuite s'ils ont mérité leur place.

La seconde règle concerne le vocabulaire : *Intent*, *Status/State* et *Event* ne sont nommés qu'après avoir été utilisés. Ils apparaissent vers l'étape 42, pas avant.

L'itinéraire prévu, dès l'étape 0 :

```
direct method calls → egoistic services → mediators → separate processes
→ filesystem as transport → Syncthing → "why all these mediators?" → rewind
→ services subscribe themselves → dynamic wiring → across processes
→ across Syncthing → when a mediator IS useful → MQTT
```

1.1 Les trois questions

La formulation des trois questions compte, et il est facile de la remplacer par une version plus faible :

Telle qu'elle est posée ici	Une version plus faible
Qui **connaît** qui ?	qui *communique* avec qui
Qui **possède** quoi ?	quelles *données* détient chaque partie
Qui **décide** du câblage ?	quelle partie *relie* le tout

« Connaître » n'est pas « communiquer » : deux composants peuvent échanger des messages toute la journée sans que l'un nomme l'autre. « Décider du câblage » n'est pas « relier » : la partie qui transmet les messages n'est pas forcément celle qui a choisi la route. Les deux distinctions deviennent concrètes plus tard — la première dès l'étape 1, la seconde dans la partie III.

2 Partie I — Commencer par ce que l'on connaît déjà

Les étapes 1 à 10 restent dans une seule JVM. Le système est construit trois fois : en appels directs, en trois capacités qui refusent de se connaître, puis en ces mêmes capacités reliées par des médiateurs. Rien n'est encore distribué, et c'est bien l'idée : la seule chose qui change est qui connaît qui.

2.1 Étape 1 — Au plus simple : appeler, point

`ProceduralRot13.java` est le système entier dans un seul fichier source compact : une boucle qui lit une ligne, et une instruction qui affiche le texte transformé.

```
void main() {
    while (true) {
        var text = IO.readln();

        if (text == null)
            return;

        IO.println(rot13(text));
    }
}
```

`IO.readln()` occupe sa propre ligne, mais `rot13(text)` n'est pas une étape distincte : il est appelé *dans* l'argument de `IO.println`. Ni `rot13` ni `IO.println` ne sait quoi que ce soit de l'autre. Celui qui connaît les deux, c'est `main()`, qui les nomme et fixe leur ordre.

Ce que `main()` sait de `rot13` : qu'il existe, comment l'invoquer (un simple nom, tous deux vivent dans la même classe générée implicitement), quel argument il prend, ce qu'il retourne, qu'il est disponible *à l'instant*, et que l'exécution continue après son retour. Il ne sait pas comment `rot13` est implémenté — c'est la seule chose qu'il peut ignorer.

La garantie de disponibilité mérite d'être nommée, car elle disparaîtra plus tard : `main()` et `rot13` sont deux méthodes d'une classe déjà chargée, appelées sur le même fil, dans le même processus. Rien n'a besoin d'être démarré d'abord, aucun message ne voyage, aucun auditeur n'a besoin d'être abonné.

Observé : taper une ligne l'affiche transformée ; `Ctrl+D` termine le programme sans rien afficher, car `IO.readln()` retourne `null` et `main()` retourne.



Figure 2.1: La version procédurale. Le câblage est écrit dans le code : `main()` nomme les trois parties et fixe leur ordre.

2.2 Étape 2 — Trois capacités égoïstes

Un composant **égoïste** dit : je sais ce que je peux faire, quelles données j'accepte et quelles données je produis — et je préfère ne pas connaître le processus métier qui m'entoure. Concrètement, `ConsoleIn` ne doit pas contenir `rot13(...)`, et `ROT13` ne doit pas contenir `consoleOut(...)`.

Supprimer ces appels crée aussitôt un problème, qui se pose en question : si personne ne connaît personne, **qui fait de cela un système ?** Notez ce que la question ne dit pas : la connaissance n'a pas disparu, elle doit aller ailleurs. Il manque deux liaisons, pas une — `ConsoleIn` ↔ `ROT13` et `ROT13` ↔ `ConsoleOut`.

2.3 Étape 3 — Donner une boîte aux lettres à chaque capacité

Le plus petit mécanisme pour les liaisons manquantes : un message et une file nommée.

```
record Message(String type, String value) {}

Map<String, BlockingQueue<Message>> mailboxes = new ConcurrentHashMap<>();

BlockingQueue<Message> mailbox(String name) {
    return mailboxes.computeIfAbsent(name, _ -> new LinkedBlockingQueue<>());
}
```

Ce sont des **boîtes aux lettres** (*mailboxes*) et non des sujets (*topics*), et la distinction mérite qu'on y insiste : si deux lecteurs appellent `take()` sur la même boîte, ils sont en concurrence — chaque message va à *un seul* d'entre eux et n'est pas copié aux deux. Un *topic*, en Publish/Subscribe, remettrait à chaque abonné sa propre copie. Un lecteur par message suffit pour l'instant, et la limitation est conservée volontairement : c'est elle qui rendra visible plus tard ce que Publish/Subscribe apporte réellement.

2.4 Étapes 4–6 — Les trois capacités deviennent égoïstes

Chaque capacité ne touche plus que ses propres noms de boîtes :

Capacité	lit	écrit
ConsoleIn	le clavier	`consoleIn.outbox`
ROT13	`rot13.inbox`	`rot13.outbox`
ConsoleOut	`consoleOut.inbox`	la console

Aucune capacité ne contient le nom d'une autre, et c'est exactement le résultat : trois capacités bien isolées qui ne font absolument rien ensemble. Rien ne déplace un message de `consoleIn.outbox` vers `rot13.inbox`, ni de `rot13.outbox` vers `consoleOut.inbox`.

2.5 Étape 7 — Entre le médiateur

La colle la plus simple possible : une boucle qui retire d'une boîte et dépose dans une autre, sans rien savoir de ce qui la traverse.

```
void mediate(String from, String to) {
    while (true)
        mailbox(to).put(mailbox(from).take());
}
```

`main()` démarre cinq threads virtuels : les trois capacités, plus deux médiateurs dont les routes sont passées en arguments — `consoleIn.outbox` → `rot13.inbox` et `rot13.outbox` → `consoleOut.inbox`.

Observé : `Hello World` en entrée, `Uryyb Jbeyq` en sortie, exactement comme à l'étape 1. Mais `Ctrl+D` ne termine plus le programme. Le thread de ConsoleIn *retourne* (il ne boucle pas), et les quatre autres threads restent bloqués dans `take()` sur des boîtes vides, sans consommer de CPU. Ce qui maintient réellement le processus en vie, c'est la dernière ligne de `main()`, `Thread.currentThread().join()` : le thread principal s'attend lui-même, ce qui ne peut jamais aboutir. Les threads virtuels sont des threads *daemon* : ces quatre threads en attente n'auraient donc pas maintenu la JVM en vie à eux seuls.

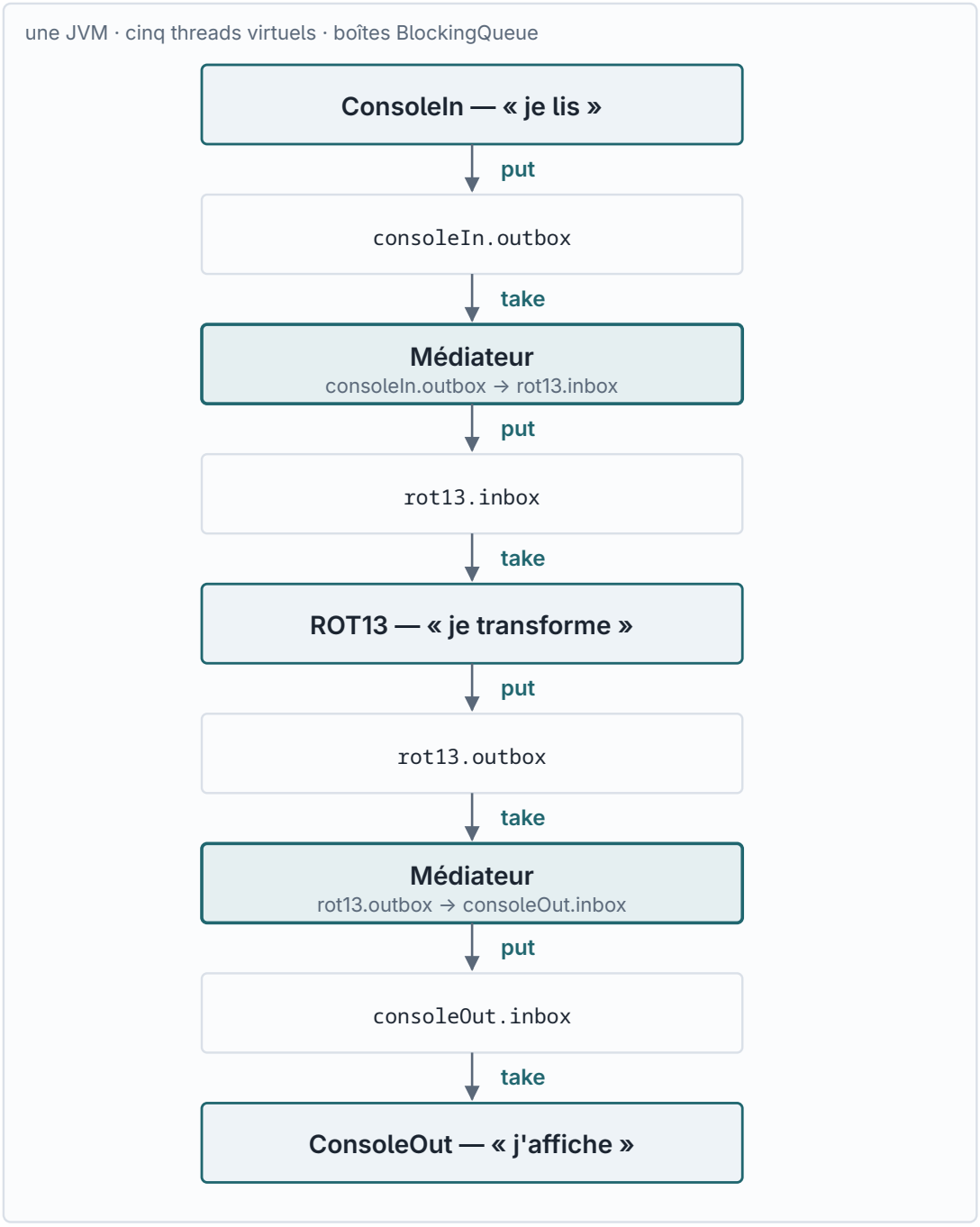


Figure 2.2: La version médiatisée dans une seule JVM. Dans la version web, **Lancer** fait parcourir à un message les huit sauts.

2.6 Étape 8 — Redemander : qui connaît qui ?

ConsoleIn connaît-il ROT13 ? Non. ROT13 connaît-il ConsoleIn ou ConsoleOut ? Non. La topologie a-t-elle disparu ? Non — elle se trouve dans les deux appels `mediate(...)` à l'intérieur de `main()`.

C'est le premier résultat majeur :

Le découplage ne fait souvent pas disparaître la connaissance. Il la déplace là où elle appartient.

La capacité « ROT13 transforme du texte » appartient à ROT13. La topologie `ConsoleIn` → `ROT13` → `ConsoleOut` appartient à la composition du système. Ce qui a changé depuis l'étape 1, ce n'est pas que le câblage a quitté `main()` — il y est toujours — mais qu'il n'est plus écrit *à l'intérieur* d'une capacité. C'est désormais une donnée : deux paires de noms de boîtes. Recâbler touche ces deux lignes et aucune capacité.

2.7 Étape 9 — Pourquoi des threads virtuels ?

Cinq boucles attendent désormais indéfiniment : le clavier, `rot13.inbox`, `consoleOut.inbox` et les sources des deux médiateurs. Les threads virtuels sont simplement le moyen Java peu coûteux de faire attendre cinq choses à la fois dans une seule JVM.

C'est une commodité d'implémentation, pas une architecture. L'architecture — qui connaît qui, et où vit le câblage — réside dans les noms de boîtes, pas dans les threads. La preuve de cette séparation : la même architecture tournera en processus séparés à l'étape 11, sans aucun thread partagé entre les parties.

2.8 Étape 10 — Garder les services égoïstes

Le raccourci évident : le premier médiateur ne fait que recopier les messages tels quels, alors pourquoi ne pas le supprimer et laisser ROT13 faire `take()` directement sur `consoleIn.outbox` ?

Parce que ROT13 contiendrait alors le nom `consoleIn.outbox`, c'est-à-dire qu'il saurait que ConsoleIn existe et où il publie. La topologie entrerait *dans* ROT13, et ROT13 cesserait d'être une capacité égoïste. Le test qui rend cela concret : supposons que l'entrée vienne d'une nouvelle capacité `FileIn` publiant dans `fileIn.outbox`.

	avec le médiateur	avec le raccourci
ce qui change	un argument du médiateur	`runRot13` lui-même
qui est touché	aucune capacité	ROT13, dont le travail n'a pas changé

Le prix mérite d'être dit clairement : ce design peut sembler indirect — voire un peu bizarre. Il existe une autre manière de garder les services égoïstes *sans* médiateur sur le chemin de chaque message ; elle viendra plus tard.

2.9 Ce que la partie I a déplacé

étape	où vit la topologie	qui peut recâbler
étape 1	dans le code de la capacité (`IO.println(rot13(text))`)	celui qui édite ce code
étapes 7–8	dans `main()`, en paires de noms de boîtes	celui qui édite `main()`

Fonctionnalité : inchangée. Lignes de code : davantage. Parties qui se connaissent : aucune. C'est le compromis que tout ce qui suit continue de mettre à l'épreuve.

3 Partie II — Franchir la frontière de processus

Les étapes 11 à 18 reprennent l'architecture de la partie I et la répartissent sur cinq processus du système d'exploitation, sur un seul ordinateur. Les capacités gardent leurs noms de boîtes ; seul le transport en dessous change, des objets Java vers des fichiers dans des répertoires. Le premier problème réellement distribué apparaît ici, et il n'a rien à voir avec ROT13.

3.1 Étape 11 — Séparer les composants

Trois processus ne peuvent pas partager une `BlockingQueue` : chaque commande `java` démarre sa propre JVM avec sa propre mémoire, la carte `mailboxes` existerait donc en trois exemplaires et chacun ne verrait que ses propres messages. Il faut un transport externe.

TCP conviendrait, MQTT conviendrait. C'est délibérément l'option la plus primitive qui est choisie, le **système de fichiers**, pour une raison en une ligne : *parce qu'on peut le voir*. Les messages deviennent des fichiers que l'on peut lister avec `ls` et lire avec `cat` pendant que le système tourne. Primitif ne veut pas dire inférieur : la partie VIII revient sur ce que le système de fichiers fait vraiment bien et vraiment mal.

3.2 Étapes 12–13 — Un répertoire devient une boîte aux lettres

La correspondance tient en une ligne :

```
directory = mailbox        file = message
publish   = create file    receive = watch directory
```

```
messageBoard/
├── consoleIn/outbox/
├── rot13/inbox/
├── rot13/outbox/
└── consoleOut/inbox/
```

Le record `Message` gagne un horodatage optionnel, et le format de fichier reste deux lignes plus une troisième facultative :

```
TEXT
Hello World
TIMESTAMP: 2026-09-22T10:42:12.123Z
```

Un message écrit à la main peut omettre l'horodatage. Le marqueur `TIMESTAMP:` empêche qu'une ligne de contenu ressemblant à une date soit prise pour une métadonnée ; dans ce format minuscule, une dernière ligne commençant par `TIMESTAMP:` est réservée.

Jackson pourrait être introduit ici et ne l'est volontairement pas. L'intérêt d'un format ridiculement simple, c'est que presque rien ne cache l'architecture — le message entier est lisible dans n'importe quel éditeur de texte, ce qui maintient l'attention sur qui parle à qui.

3.3 Étape 14 — Le premier problème de système distribué : la publication

Un lecteur qui surveille un répertoire peut découvrir un fichier *avant que l'écrivain ait fini de l'écrire*. D'où un protocole de publication :

1. créer un fichier temporaire privé ;
2. écrire le message complet ;
3. le fermer ;
4. le renommer atomiquement vers son nom final `.msg` ;
5. les consommateurs ne réagissent qu'aux fichiers finaux.

```
Files.writeString(temporary, content);
Files.move(temporary, published, StandardCopyOption.ATOMIC_MOVE);
```

Les deux moitiés sont nécessaires. Le renommage est atomique — un lecteur voit soit aucun fichier `.msg`, soit le fichier complet — mais il ne protège rien si les lecteurs ramassent aussi les `.tmp`. `.tmp` et `.msg` restent dans le même répertoire, pour que le déplacement ne traverse pas de système de fichiers, où l'atomicité serait perdue.

> **Écrire et publier ne sont pas la même opération.**

3.4 Étape 15 — Observer une boîte avec `WatchService`

L'observateur de système de fichiers de Java fournit la moitié consommateur : enregistrer un répertoire pour `ENTRY_CREATE`, se bloquer dans `watcher.take()`, et ignorer tout nom de fichier qui ne finit pas par `.msg`.

Cela donne aussi un outil d'injection très rudimentaire. Avec `ActuatorService` lancé comme consommateur :

```
printf 'TEXT\nHello from a file' > messageBoard/consoleOut/inbox/manual.tmp
mv messageBoard/consoleOut/inbox/manual.tmp messageBoard/consoleOut/inbox/manual.msg
```

Observé : rien ne se passe à l'apparition du fichier `.tmp` ; la ligne est affichée à l'instant où le renommage crée le nom `.msg`. Les deux formats acceptés fonctionnent, avec et sans horodatage final.

Le point architectural s'écrit `producer` → `filesystem` → `consumer` : le consommateur n'a pas pu distinguer un humain tapant `mv` d'un véritable service. Il connaît sa propre boîte d'entrée, pas qui y écrit. L'égoïsme a survécu à la frontière de processus.

3.5 Étapes 16–17 — Services et médiateur deviennent des processus

Les trois capacités deviennent trois programmes, et le médiateur un quatrième type de programme :

programme	joue	lit	écrit
`SensorService.java`	ConsoleIn	le clavier	`consoleIn/outbox`
`ProcessorService.java`	ROT13	`rot13/inbox`	`rot13/outbox`
`ActuatorService.java`	ConsoleOut	`consoleOut/inbox`	la console

programme	joue	lit	écrit
`Mediator.java <from> <to>`	la colle	`<from>`	`<to>`

Le médiateur est générique : sa route n'est pas compilée en dur, elle arrive en deux arguments de démarrage. Mais une asymétrie mérite d'être gardée en tête : les trois capacités reçoivent leur **travail** sous forme de messages dans une boîte, tandis que le médiateur reçoit sa **route** comme configuration de démarrage, et il n'a ni nom ni boîte de contrôle propre où quelqu'un pourrait écrire.

Conséquence pratique : les arguments sont lus une seule fois, donc changer la route d'un médiateur en cours d'exécution exige de l'arrêter et de le relancer. Un fichier de configuration ne changerait rien — il est lui aussi lu au démarrage. Ce point reste volontairement ouvert jusqu'à ce que le coût des médiateurs distribués ait été ressenti.

3.6 Étape 18 — Exécuter la version entièrement externe, en local

Cinq terminaux, cinq processus, un seul système de fichiers local entre eux :

```
java SensorService.java                                  # terminal 1
java Mediator.java consoleIn/outbox rot13/inbox          # terminal 2
java ProcessorService.java                               # terminal 3
java Mediator.java rot13/outbox consoleOut/inbox         # terminal 4
java ActuatorService.java                                # terminal 5
```

Observé : `Hello World` tapé dans le terminal 1 est apparu comme `Uryyb Jbeyq` dans le terminal 5. Les trois terminaux du milieu n'ont rien affiché de nouveau — chaque médiateur affiche une ligne « Mediating: … » au démarrage, et `ProcessorService` n'affiche rien du tout. Ensuite, chacun des quatre répertoires de boîtes contenait un fichier `.msg` : la même ligne, recopiée le long de la chaîne.

Comptons : **cinq processus** pour ce que l'étape 1 faisait en une instruction. Ce n'est pas le moment d'optimiser : ce n'est pas encore censé être confortable.

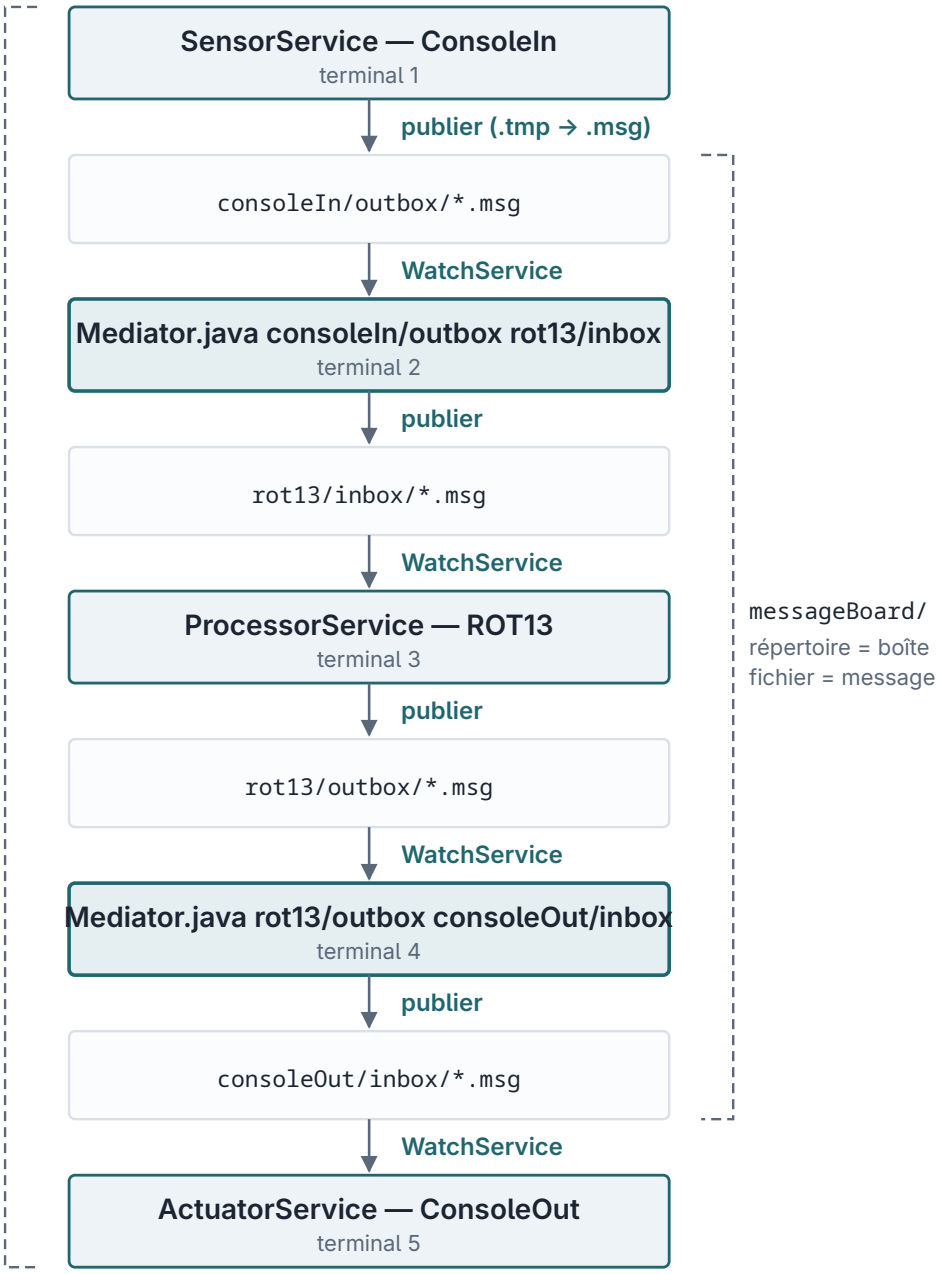


Figure 3.1: Les cinq processus et les quatre répertoires de boîtes. Dans la version web, **Lancer** suit une ligne de texte du clavier à la console.

3.7 Ce que la partie II a déplacé

étape	où vit la topologie	qui peut recâbler
étape 1	dans le code de la capacité	celui qui édite ce code

étape	où vit la topologie	qui peut recâbler
étapes 7–8	dans `main()`, en noms de boîtes	celui qui édite `main()`
étape 18	dans les deux lignes de commande `Mediator.java <from> <to>`	celui qui lance les processus

La topologie a maintenant quitté le code source : aucun fichier `.java` ne contient la route. Elle vit dans ce que l'opérateur a tapé, ce qui fait d'une personne l'orchestrateur actuel. Ce que la version à cinq processus apporte est réel — le câblage peut changer sans toucher aucun programme — et ce qu'elle coûte l'est tout autant : cinq processus, quatre copies de chaque message, et un programme qui ne s'arrête plus quand on appuie sur `Ctrl+D`.

4 Partie III — Le rendre réellement distribué

Cinq processus sur un seul ordinateur partagent encore un même système de fichiers. Les étapes 19 à 27 les répartissent sur deux machines, répliquent le tableau de messages entre elles avec Syncthing, mesurent ce que cela coûte, puis cassent délibérément un médiateur. À la fin, le prix des médiateurs est un chiffre, non une opinion.

4.1 Syncthing réplique ; il ne distribue pas

Syncthing maintient un répertoire répliqué entre plusieurs machines. C'est tout ce qu'on lui demande ici. Il ne sait rien d'Intent, de Status, d'Event, de ROT13 ni des médiateurs, et il n'a aucune notion de consommateur : pas d'abonnement, pas de sujet, aucune garantie de livraison, aucune idée que quelqu'un attend.

Le déclencheur reste le nôtre. Chaque service exécute son propre `WatchService` sur un répertoire **local** ; Syncthing fait seulement apparaître les fichiers à cet endroit. C'est pourquoi aucun des trois services n'a changé d'une ligne lorsque le système est devenu distribué.

Deux conséquences en découlent, et toutes deux apparaîtront comme des observations plutôt que comme des avertissements :

- la réplication est **asynchrone** — le renommage `.tmp` → `.msg` est atomique sur chaque machine séparément, et rien ne promet d'ordre ni d'atomicité entre machines ;
- le système de fichiers est un transport sérieux, avec ses forces : les fichiers peuvent être gros, les répliques survivent à un redémarrage, et chaque message se lit avec `cat`.

4.2 Tester une couche à la fois

Avant que le moindre programme Java ne touche au dossier répliqué, le test le plus ennuyeux possible a été fait d'abord : écrire `hello.txt` sur une machine, vérifier qu'il apparaît sur l'autre, le supprimer, vérifier que la suppression revient. Les deux sens fonctionnent.

Cet ordre compte. Quand l'exécution distribuée a échoué plus tard, la question « est-ce la réplication ou mon programme ? » était déjà tranchée pour le transport, et la recherche pouvait commencer une couche plus haut.

4.3 Ce que coûte une frappe

Avec le capteur sur une machine et les médiateurs, ROT13 et l'actionneur sur l'autre, une ligne tapée a produit **quatre** fichiers `.msg`, un par boîte :

boîte	contenu	horodatage interne
`consoleIn/outbox`	`Hello World`	23:02:05.315
`rot13/inbox`	`Hello World`	23:02:05.315

boîte	contenu	horodatage interne
`rot13/outbox`	`Uryyb Jbeyq`	23:02:13.471
`consoleOut/inbox`	`Uryyb Jbeyq`	23:02:13.471

Les horodatages répétés prouvent que deux de ces quatre publications sont de **pures copies** : les médiateurs republient le message inchangé, horodatage compris. Ils calculent `identity(x) = x`, sous forme de processus distribués distincts.

Le temps, mesuré sur les dates de modification des quatre fichiers d'une seule machine pour qu'aucun décalage d'horloge n'intervienne : le fichier du capteur apparaît à `01:02:05.432`, les trois autres à `01:02:13.460`, `.504` et `.532` — **8,10 s** plus tard, et à **72 ms** l'un de l'autre. Le travail propre de la chaîne a donc pris des millisecondes ; les huit secondes sont la propagation de Syncthing. La même chaîne, exécutée entièrement sur une machine, termine en 24 ms environ.

4.4 Casser un médiateur

Arrêtez le premier médiateur et tapez deux lignes. ConsoleIn est vivant, ROT13 est vivant, ConsoleOut est vivant, Syncthing est vivant — et le flux métier est mort. Les deux lignes s'accumulent dans `consoleIn/outbox` ; rien ne va plus loin. Les médiateurs font partie de la chaîne de disponibilité.

Redémarrez ensuite le médiateur. **Rien n'est livré.** `FileMessageBoard.watch` enregistre un répertoire pour `ENTRY_CREATE` et se bloque dans `watcher.take()` ; il ne liste jamais ce qui est déjà là. Les deux fichiers sont toujours sur le disque, dans les deux répliques : aucune donnée n'a été perdue. Ce qui n'a pas de mémoire, c'est l'*abonnement*.

La distinction mérite d'être retenue : **les messages ont survécu, la livraison non.** Un courtier répond à cela par l'état de session et les messages retenus ; dans notre propre code, ce serait une ligne — lister les fichiers `.msg` existants avant d'entrer dans la boucle d'observation — que le matériel se garde volontairement d'écrire pour l'instant.

4.5 Le médiateur devient un service ordinaire

Avant de supprimer les médiateurs, une asymétrie disparaît : les services reçoivent leur *travail* sous forme de messages, alors que le médiateur recevait sa *route* en arguments de shell. On ne lui donne donc qu'un nom, `ConfigurableMediator mediator1`, dont il dérive une boîte de contrôle permanente, `mediators/mediator1/inbox`, et la route arrive comme message :

```
ROUTE
consoleIn/outbox
rot13/inbox
```

`from` et `to` voyagent dans **un seul** message, volontairement : en deux, le médiateur combinerait brièvement une nouvelle source avec l'ancienne destination et transmettrait au mauvais endroit. Une route est un seul fait, et un fait tient dans un message.

Un orchestrateur éphémère publie les deux routes puis s'arrête : la topologie est visible en un seul endroit et n'est **pas** dans le chemin des données. Le lancer après les services, en ayant tapé une ligne avant, montre les deux moitiés d'un coup : la ligne précoce reste bloquée, la suivante circule.

Ce qui s'améliore : le médiateur a une identité, reçoit sa configuration par sa propre boîte, garde une capacité générique et tient sa topologie d'un orchestrateur. Ce qui ne s'améliore pas : le chemin des

données. Il reste `event` → `médiateur` → `copie de l'event`, toujours un processus dans la chaîne de disponibilité, qui copie chaque message sans rien y ajouter. Et cette version accepte exactement **une** route : `waitForRoute(...)` retourne, puis le médiateur entre dans `watch(route.from(), …)` et ne lit plus jamais sa boîte de contrôle. Le recâblage à chaud n'est pas promis, car un `watch` bloquant ne peut pas servir à la fois une boîte de contrôle et une source — d'où l'aide à l'abonnement qui apparaît plus tard.

4.6 Ce que la partie III a déplacé

étape	où vit la topologie	dans le chemin des données ?
étape 18	les lignes de commande de l'opérateur	les médiateurs y sont
étape 27	un orchestrateur, publiée comme Intent	les médiateurs y sont encore

La question sur laquelle se referme la partie est donc précise et non rhétorique : **pourquoi le service intéressé par la donnée ne pourrait-il pas l'écouter lui-même ?**

5 Partie IV — Rembobinage

Les étapes 28 à 34 reviennent volontairement à une seule JVM. Sans délai de réplication ni cinq terminaux, la seule chose qui peut changer est qui connaît qui — l'idée s'essaie donc en quelques secondes plutôt que pendant une séance de débogage distribué.

5.1 Le raccourci évident, et son piège

Si le premier médiateur ne fait que copier des messages, supprimons-le et laissons ROT13 lire la source lui-même :

```
listen("consoleIn.outbox");
```

Le médiateur a disparu. Mais posons la question récurrente : ROT13 contient désormais le nom de la boîte de ConsoleIn. La topologie a quitté le médiateur pour entrer **dans le service**. Comme le dit le matériel : « nous avons supprimé le médiateur… et remis la topologie dans le service. *Oups.* »

5.2 La capacité n'est pas la topologie

ROT13 peut savoir deux choses : *je sais transformer du texte* et *je suis capable d'écouter des messages texte compatibles*. Il ne doit **pas** savoir *mon entrée vient toujours de consoleIn.outbox* — c'est de la topologie de déploiement.

Le nom n'est donc pas compilé ; il arrive à l'exécution :

```
TOPIC consoleIn.outbox
```

ROT13 apprend son câblage par la donnée. Notez qui l'envoie : dans `DynamicQueueRot13.java`, c'est `main()` qui publie les deux messages `TOPIC` — un configurateur, hors du chemin des données, dans le rôle qu'occupait l'orchestrateur. Ce ne peut pas être ConsoleIn : s'il disait à ROT13 à quoi s'abonner, il devrait savoir que ROT13 existe, et la topologie aurait seulement changé de capacité d'accueil.

5.3 Ce qui change dans le code

`listen(...)` n'a rien d'exotique : un thread virtuel qui appelle `take()` en boucle sur l'une des mêmes files. `subscribe(...)` démarre un tel thread par nom abonné. La conséquence se voit dans les boîtes :

- ROT13 retire **directement de `consoleIn.outbox`** — le nom qu'on lui a transmis ;
- `rot13.inbox` n'est plus une boîte de charge utile. Elle porte des messages de contrôle, et la boucle de ROT13 la lit pour décider si un message est un `TOPIC` à appliquer.

Plus rien n'écrit dans `rot13.inbox` pour déplacer des données. La boîte n'a pas disparu : elle a changé de métier, de la charge utile vers l'Intent.

Une observation à l'exécution : après `Ctrl+D`, la sortie s'arrête mais le programme reste en vie. Rien n'a été interrompu — `runConsoleIn` retourne à la fin de l'entrée, donc le **producteur** a disparu, tandis que les deux threads abonnés restent bloqués dans `take()` sur des sources qui ne publieront plus. La JVM survit parce que `main` finit par `Thread.currentThread().join()`. « Le flux s'est arrêté » doit toujours être ramené à *quel* participant s'est arrêté.

5.4 `ROUTE` et `TOPIC` sont de l'Intent

Appliquons deux tests à `TOPIC consoleIn.outbox`. Décrit-il quelque chose qui est arrivé ? Non. Quelque chose actuellement vrai ? Non. Il dit *commence à écouter ici* — un comportement souhaité, donc de l'**Intent**. Le `ROUTE` du médiateur est du même genre : *transmets d'ici à là*.

Un piège à nommer, car il est tentant dans une soutenance : *publier* un Intent est bien sûr quelque chose qui est arrivé. La classification suit ce que dit le message, pas le fait qu'il existe.

Et une condition, qui reviendra : la version dynamique n'est **pas** automatiquement supérieure. Elle n'a de sens que si le consommateur comprend vraiment le contrat de message du producteur.

```
MÉDIÉ STATIQUE                        DIRECT DYNAMIQUE
Producteur → Médiateur → Consommateur   Intent → (Producteur ⟶ Consommateur)
```

6 Partie V — Rejouer l'idée dynamique entre processus

Les étapes 35 à 38 sortent l'idée d'abonnement de la JVM unique. Les services redeviennent des programmes séparés, mais au lieu de médiateurs qui copient des fichiers, chaque service observe la boîte qu'on lui a *indiquée*. Une hypothèse du premier utilitaire de fichiers ne survit pas au déplacement, et c'est bien de la corriger qu'il s'agit ici.

6.1 Une observation bloquante ne suffit pas

`FileMessageBoard.watch(topic, receiver)` enregistre **un** répertoire et se bloque indéfiniment. Cela suffisait tant que chaque processus observait une boîte fixe. Un service câblé dynamiquement demande davantage :

```
observer en permanence sa propre boîte de contrôle
ET
ajouter un répertoire observé à l'exécution
ET
éventuellement le retirer ensuite
```

L'ancienne signature ne peut pas exprimer cela, et la conséquence était déjà visible : le médiateur configurable n'acceptait qu'une seule `ROUTE`, car dès qu'il observait sa source il devenait sourd à sa propre boîte de contrôle. La limite n'a jamais été *de qui* est le répertoire, mais *un seul, fixe, pour toujours*.

6.2 L'utilitaire : `FileSubscriptions`

Un seul `WatchService` peut servir plusieurs répertoires. `register(...)` retourne un `WatchKey` qui en identifie un, `watcher.take()` retourne la clé qui a des événements, et `key.cancel()` la désenregistre. L'utilitaire garde les deux sens, `nom` → `clé` et `clé` → `nom`, pour qu'un événement puisse être rattaché à la boîte dont il vient.

Les événements n'arrivent pas directement dans un callback. Ils passent par une file, lue avec `take()`, et portent

```
record IncomingMessage(String mailbox, Message message) {}
```

Deux raisons, l'ordre d'arrivée n'étant que la troisième. D'abord, le thread observateur doit rester réactif pour *tous* les répertoires abonnés : un callback qui transforme, publie ou bloque arrêterait l'observation de tous les autres. Ensuite, le nom de la boîte voyage avec le message, de sorte qu'**une seule** boucle du service peut décider de son sens — boîte de contrôle, donc un Intent à appliquer ; source abonnée, donc des données à transformer — et cette même boucle peut ajouter ou retirer des abonnements, puisqu'elle n'est pas prisonnière d'un callback.

6.3 Deux publications au lieu de quatre

Lancez `SensorService`, `DynamicProcessorService` et `DynamicActuatorService` — sans médiateurs — puis laissez `Configure.java` publier les deux messages `TOPIC`. Une ligne tapée ensuite laisse exactement deux publications applicatives dans le tableau :

boîte	contenu	nature
`consoleIn/outbox`	`TEXT hello`	l'entrée
`rot13/outbox`	`TEXT uryyb`	le résultat
`rot13/inbox`	`TOPIC consoleIn/outbox`	Intent, publié une fois
`consoleOut/inbox`	`TOPIC rot13/outbox`	Intent, publié une fois

`consoleOut/inbox` ne contient **aucune charge utile** : ConsoleOut retire directement de `rot13/outbox`. Sa boîte a changé de métier, des messages vers le câblage.

Une ligne tapée *avant* l'exécution de `Configure.java` reste dans `consoleIn/outbox` pour toujours. Un abonnement qui n'existe pas encore manque ce qui l'a précédé — le même constat que le médiateur redémarré qui n'a rien livré.

version	publications applicatives	processus dans le chemin des données
médiée (étape 18)	4	5
médiateur configurable (étape 27)	4	5
dynamique (étape 38)	**2**	**3**

7 Partie VI — Rejouer sur Syncthing, et nommer les messages

Les étapes 39 à 47 répartissent la version dynamique sur deux machines, précisent ce que « direct » veut dire, puis — maintenant seulement, après avoir utilisé les idées — donnent leurs noms aux trois sortes de messages : Intent, Status, Event.

7.1 L'exécution dynamique distribuée

Capteur sur une machine, ROT13 et l'actionneur sur l'autre, aucun médiateur. Après la réplication des deux Intents `TOPIC`, une ligne tapée a produit le résultat sur l'autre machine — et le tableau montre ce que cela coûte :

- `consoleIn/outbox` : l'entrée ;
- `rot13/outbox` : le résultat, écrit par le service qui l'a produit ;
- `consoleOut/inbox` : **de l'Intent seulement**, là où la version médiée contenait une copie complète de chaque message.

Le temps est la partie intéressante. L'entrée a été publiée à 02:35:12 et les résultats sont apparus sur la première machine à 04:11:28,797 et ,893 — à 96 ms d'écart, bien après l'entrée. Syncthing était saturé pendant la nuit : le lot est arrivé quand la file s'est vidée. La chaîne est restée correcte ; la latence du transport n'est tout simplement pas contrôlée par l'architecture. À comparer aux 24 ms de la même chaîne sur une seule machine.

7.2 Ce que « direct » veut dire, exactement

Deux sauts qui se ressemblent sont architecturalement opposés :

- un **saut de transport** — Syncthing de chaque côté, éventuellement un relais entre les deux — n'ajoute aucun participant qui connaisse la topologie, et rien n'y réagit. Si les paquets passaient par un autre continent, aucune affirmation ici ne serait contredite.
- un **saut architectural** — un médiateur copiant `consoleIn/outbox` → `rot13/inbox` — ajoute un composant qui doit connaître les deux noms, se place dans la chaîne de disponibilité et republie le message.

L'affirmation défendable est donc : *ROT13 réagit à la publication de ConsoleIn sans aucun de nos composants entre les deux, et sans contenir l'identité de ConsoleIn.* Et non : *les données vont directement de A à B.* L'architecture décide à quels messages un service réagit, pas du routage du transport.

7.3 Intent, Status, Event

Trois sortes de messages, nommées seulement après avoir été utilisées :

	dit	nos exemples
E — Event	quelque chose **est arrivé**	`TEXT hello` comme `ConsoleIn/E/text` ; `TEXT uryyb` comme `ROT13/E/crypted`
S — Status/State	quelque chose est **actuellement vrai**	`ROT13/S/subscriptions` — pas encore publié par notre système
I — Intent	ce qui **devrait** arriver	`TOPIC consoleIn/outbox`, `ROUTE … …`

Le passé composé est un bon indice pour les Events. Pour l'Intent, le test tient en deux questions : le message rapporte-t-il quelque chose qui est arrivé ? Non. Quelque chose d'actuellement vrai ? Non. Alors il demande un comportement.

Un piège : *publier* un Intent est bien sûr quelque chose qui est arrivé. Il faut classer d'après ce que dit le message, non d'après son existence — sinon tout devient Event et le vocabulaire ne sert plus à rien.

Le Status mérite une note, car l'instinct est de le **demander** : « quels sont tes abonnements ? » Mais demander, c'est appeler, et un appel exige que le demandeur connaisse le composant, le sache en vie, et attende. Le Status se **publie** : ROT13 publie `ROT13/S/subscriptions` dès que son câblage change, et quiconque s'y intéresse le lit. Le lecteur a alors besoin d'un sujet, pas d'un composant, et ROT13 n'apprend jamais qui l'audite. Chaque sorte répond à une question différente :

- un Event n'est utile que si l'on écoutait — nos messages bloqués, trois fois de suite ;
- un Status est ce dont un nouveau venu a besoin : la valeur actuelle, sans rejouer l'histoire ;
- un Intent demande un changement.

Et les commandes ne sont pas interdites. Envoyer `ROT13/I` avec `ACTION Hello` ressemble à une commande, et le verdict du matériel est « Fine. » Le composant exerce sa capacité et publie ses Events et son Status. Le flux retenu est `Intent` → `composant` → `State + Events`.

7.4 La règle de l'écrivain unique

> Toute ressource écrivable a exactement une autorité d'écriture.

Ne pas confondre avec l'UUID du nom de fichier. L'UUID empêche deux publications de choisir le même nom physique ; l'autorité d'écriture dit quel composant **possède** une ressource logique d'Event ou de Status. Deux écrivains n'entrent jamais en collision sur un nom de fichier, et c'est précisément pour cela que le problème passe inaperçu.

Ce que coûte un second écrivain : le nom de la boîte était le seul marqueur de provenance, donc `rot13/outbox` cesse de signifier « ROT13 a transformé ceci » et un abonné ne peut plus faire confiance au contrat sur lequel il s'appuie. Pour le Status c'est pire — deux composants prétendant chacun publier *la valeur actuelle* d'un même fait se contredisent, sans aucune garantie d'ordre entre machines pour trancher. Pour des Events en ajout seul, de nombreuses publications coexistent par construction ; la question est la propriété sémantique, pas la collision. Quand deux autorités apparaissent, demandez pourquoi il y en a deux plutôt que de sortir un verrou.

8 Partie VII — Fusionner les deux idées architecturales

Deux médiateurs ont été supprimés et le gain mesuré. Les étapes 48 à 52 répondent à la question suivante, évidente : la médiation est-elle pour autant mauvaise ? Non — et la distinction qui tranche est assez nette pour s'appliquer à une conception qu'on n'a jamais vue.

8.1 Un transformateur mérite d'exister

Ce qui a été supprimé, c'est un médiateur calculant `A` → `A` : une fonction identité coûteuse, dans la chaîne de disponibilité, copiant chaque message sans rien y ajouter.

Prenons maintenant un producteur publiant `{"value":"Hello"}` et un consommateur attendant une structure d'affichage. Un `TextToDisplayTransformer` entre les deux apporte transformation de schéma, adaptation sémantique, éventuellement conversion d'unités, éventuellement agrégation. C'est un travail qu'aucun des deux ne devrait porter : ROT13 n'a pas à décider des couleurs du terminal, et la console n'a pas à comprendre la forme de la charge utile de ROT13.

La ligne se trace par la **compatibilité des contrats**, non par le fait d'ajouter quelque chose. Un médiateur convertissant des Celsius en Fahrenheit modifie au lieu d'ajouter et mérite sa place ; un médiateur ajoutant un horodatage à un message que le consommateur comprend déjà a ajouté quelque chose et reste suspect. La question est toujours : le consommateur pourrait-il lire le message du producteur tel quel ?

> Si un consommateur comprend sémantiquement la publication d'un producteur, l'abonnement direct suffit souvent. Si les contrats sont incompatibles, introduire un transformateur qui effectue une adaptation réelle.
>
> Un médiateur qui ne fait que transmettre des données compatibles est suspect. Un médiateur qui ajoute la sémantique nécessaire mérite d'exister.

8.2 Le transformateur est égoïste aussi

Inséré dans la chaîne, le transformateur s'abonne à `rot13/outbox` et publie dans **sa propre** boîte de sortie ; ConsoleOut s'abonne alors à celle du transformateur plutôt qu'à celle de ROT13. Notez ce que cela coûte en code : **rien**. ConsoleOut reçoit un autre Intent — `TOPIC display/outbox` au lieu de `TOPIC rot13/outbox` — et la chaîne est recâblée. ROT13 n'apprend jamais que le transformateur existe ; le transformateur n'apprend jamais que ConsoleOut existe.

Pourquoi ne pas publier directement dans `consoleOut/inbox`, en économisant une boîte ? Quatre raisons :

- **connaissance** — il contiendrait le nom de ConsoleOut, c'est-à-dire de la topologie dans un service, la rechute de l'étape 28 ;
- **propriété** — cette boîte est le canal de contrôle de ConsoleOut, écrit par le configurateur ; une charge utile venue du transformateur lui donnerait deux autorités ;

- **portée** — sa propre boîte peut être lue par un journal, une vue web ou un second affichage sans frais, alors qu'une boîte d'entrée ne sert qu'un consommateur ;
- **mutabilité** — un agrégateur peut être inséré ensuite en envoyant un autre Intent, sans modifier aucun composant.

8.3 L'architecture complète sur système de fichiers

Chaque boîte avec exactement un écrivain :

boîte	écrivain unique
`consoleIn/outbox`	SensorService
`rot13/outbox`	ProcessorService
`display/outbox`	le transformateur
chaque boîte de contrôle	le configurateur — l'unique autorité d'Intent désignée
le terminal	ActuatorService, qui n'écrit aucune boîte

Des capacités égoïstes, l'Intent venant d'une seule autorité, un transformateur seulement là où les contrats diffèrent, un écrivain par ressource, et la topologie nulle part dans les capacités. C'est la conception que le système de fichiers a suffi à enseigner — et la partie suivante demande à quoi le système de fichiers, lui, ne suffit plus.

9 Partie IX — MQTT

9.1 Ce que le système de fichiers a enseigné, et ce pour quoi il n'est pas fait

Le système de fichiers a enseigné les frontières de publication, les répertoires comme adresses, l'observation, des processus indépendants, des langages indépendants, la latence de synchronisation, le câblage à l'exécution, le State, l'Intent, les Events, le coût des médiateurs, la propriété des services et les services transformateurs. Ce fut un bon professeur, et ce n'est pas un courtier de messages élégant : à la fin, l'architecture était propre et le transport était un amas de fichiers `.tmp`, de déplacements atomiques, de rescans, de chemins partagés et de délais de réplication.

La conclusion n'est **pas** « fichiers mauvais, MQTT bon ». Le système de fichiers a un avantage que MQTT ne peut égaler — il stocke couramment des gigaoctets ou des téraoctets, alors que courtiers et clients sont conçus pour de petits messages, et découper une charge énorme sur MQTT revient à inventer un protocole de transfert de fichiers dans un protocole de messages. Donc :

> Utilisez les messages pour la coordination et les événements. Utilisez un stockage de fichiers ou d'objets adapté pour les grandes données. Un message peut porter l'**emplacement, l'identité et les métadonnées** de ces données au lieu des téraoctets eux-mêmes.

Un système de fichiers répliqué donne aussi à chaque machine une copie locale, ce qui est une vraie force — mais la réplication n'est pas une sauvegarde : suppressions et corruptions se répliquent aussi, à moins qu'un versionnement conserve les copies antérieures.

9.2 MQTT devrait sembler familier

système de fichiers / Syncthing	MQTT
racine du tableau de messages synchronisé	courtier / espace de noms
chemin de répertoire	sujet (*topic*)
fichier `.msg`	publication
contenu du fichier	charge utile
`WatchService`	abonnement
répertoire dynamique	sujet / filtre
répertoire de contrôle	`I`
répertoire d'état	`S/…`
répertoire d'événements	`E/…`

La réaction visée par le matériel : « Ah. Nous connaissons déjà cette architecture. » MQTT est un changement de transport, pas une affirmation architecturale.

Deux choses qu'un répertoire ne pouvait pas faire, visibles en une minute en ligne de commande. Une publication faite **avant** l'existence d'un abonné ne lui parvient jamais — le constat des messages bloqués, pour la troisième fois, mais avec une réponse que le système de fichiers n'avait pas. Et un

filtre comme `test/#` correspond à `test/a` et `test/a/b/c` mais pas à `testing/x`, car `#` travaille par **niveaux** et non par préfixes de chaîne ; `test/+` ne correspondrait qu'à un seul niveau.

La correction plus profonde : **un sujet n'est pas un conteneur.** Il n'existe aucun objet nommé `test/a` sur le courtier. Un sujet est une clé de routage portée par chaque publication et confrontée aux filtres de qui est abonné à cet instant. Les répertoires *contiennent* les messages ; les sujets les *filtrent*. Un seul abonnement couvre donc des sujets qui n'existent pas encore — là où un tableau de messages exigeait un nouveau répertoire et un nouvel enregistrement.

9.3 La bibliothèque, l'identifiant client et l'adresse du courtier

Java n'a pas de client MQTT dans sa bibliothèque standard ; Paho arrive donc maintenant — après le besoin, comme le médiateur à l'étape 7 et Jackson à l'étape 64. Quatre choses suffisent : se connecter, publier, s'abonner, rappeler. La connexion est volontairement anticlimatique : « Rien d'architectural ne s'est produit ici. Nous avons simplement attaché notre processus au transport. »

Deux détails de la connexion comptent :

- **l'identifiant client** n'est pas qu'un jeton d'unicité. Le courtier reconnaît par lui un client qui revient et conserve son état de session : abonnements, messages en file en QoS > 0, et l'association à sa dernière volonté. Un répertoire ne savait jamais qui l'observait, et c'est précisément pour cela qu'un observateur redémarré ne rejouait rien.
- **l'adresse du courtier** est de la configuration de déploiement : chaque programme lit `System.getProperty("mqtt.broker", "tcp://localhost:1883")`. C'est aussi la seule configuration qui ne peut pas arriver par Intent, puisque l'Intent passe *par* le courtier. La topologie se publie ; l'adresse de ce qui la publie, non.

S'abonner coûte un paquet, et **le courtier détient la liste des abonnements**. Il n'existe aucun moyen standard de lui demander à quoi un autre client s'est abonné — d'où `S/subscriptions` : la déclaration par un composant de son propre câblage, publiée pour que l'architecture sache ce que le transport garde privé. Seul son propriétaire peut la publier, par la règle de l'écrivain unique, car seul lui sait ce qu'il a accepté.

9.4 Intent structuré, et deux mappeurs

`TOPIC some/path` a un champ et un sens. Il ne peut pas dire de quel *type* d'Intent il s'agit, porter un second paramètre, s'imbriquer ou s'étendre ; et un récepteur n'a rien contre quoi valider. Un Intent plus riche demande une structure :

```
subscribe:
  topic: Demo/ConsoleIn/U/alice/E/text
```

C'est seulement là qu'une bibliothèque de sérialisation a une tâche claire. `JsonTools` garde **deux** mappeurs, et l'asymétrie est voulue :

- en entrée, une fabrique **YAML** avec `FAIL_ON_UNKNOWN_PROPERTIES` désactivé — JSON étant un sous-ensemble de YAML, un Intent tapé à la main est accepté, et un configurateur plus récent peut envoyer des champs qu'un service plus ancien ignore ;
- l'écriture utilise du **JSON** strict avec `INDENT_OUTPUT` activé, donc les charges publiées sont strictes et lisibles.

Tolérant en entrée, strict en sortie. Un mappeur unique devrait renoncer à l'un des deux.

9.5 DUISE, et recâbler un système en marche

Le schéma de nommage reçoit enfin son nom : **D**escription (quels contrats ce service comprend et publie), **U**nit (quelle instance), **I**ntent (comportement demandé), **S**tatus (ce qui est vrai maintenant), **E**vent (ce qui est arrivé).

```
Demo/ROT13/U/alice/{I, S/online, S/subscriptions/text, E/crypted}
```

`U/<unit>` est ce dont le système de fichiers n'avait pas besoin : chaque machine avait sa réplique du même chemin relatif, l'identité d'instance était implicite. MQTT a un espace de noms partagé : un sujet doit donc dire *quelle* copie. Coder en dur `U/alice` dans une capacité est donc pire que de coder en dur `consoleIn.outbox` : cela lie la compilation à un déploiement. Configuré par Intent, **une seule compilation sert tout le monde** — d'où les trois noms d'unité en arguments de `MqttConfigure`. Les noms d'unité *sont* la topologie.

Recâblage observé sur un système en marche : publiez un nouvel Intent nommant une autre source, et le service se désabonne de l'ancienne, s'abonne à la nouvelle et republie `S/subscriptions` en conséquence. Sans redémarrage, sans édition. Envoyez au contraire un filtre invalide — `Demo/#/E/text`, où `#` n'est pas le dernier niveau — et `MqttTools.isValidFilter` le rejette ; l'ancien abonnement demeure et le Status continue de dire vrai.

Ce rejet révèle une vraie lacune, à nommer plutôt qu'à cacher : il n'est journalisé que localement. Le configurateur ne peut savoir si son Intent a été appliqué ou refusé, et un moniteur ne peut distinguer « refusé » de « pas encore traité ». Dans cette conception, l'émetteur apprend le résultat en lisant le Status et en le comparant à sa demande. Publier les refus comme Event, ou un Status de dernière erreur, combleraient la lacune ; ni l'un ni l'autre n'est dans le matériel : c'est une décision de conception à justifier.

9.6 Ce qu'un nouvel abonné apprend, et ce qu'un mort avoue

S'abonner à `Demo/#` pendant trois secondes, sans que rien ne soit publié, renvoie ceci :

```
Demo/ROT13/U/nitro5/S/online              {"value":true}
Demo/ROT13/U/nitro5/S/subscriptions/text  "Demo/ConsoleIn/U/nitro5/E/text"
Demo/ConsoleOut/U/nitro5/S/online         {"value":true}
Demo/ConsoleOut/U/nitro5/S/subscriptions/text  "Demo/ROT13/U/nitro5/E/crypted"
```

Un arrivant tardif apprend instantanément tout le câblage actuel du système, sans rien demander et sans qu'aucun composant sache qu'un moniteur existe. C'est le **Status** retenu. Aucun Event n'arrive, car les Events sont publiés non retenus — « State retenu, Events non retenus ». C'est aussi la bonne réponse à l'instinct de *demander* à un service ce qu'il fait : personne ne demande ; le service le déclare, une fois, et le courtier le garde pour quiconque se présente.

Le Status retenu seul mentirait pourtant. Tuez un service par `Ctrl+C` et son `S/online = true` reste sur le courtier à jamais. D'où la **dernière volonté**, enregistrée au moment de la connexion :

```
var will = new MqttMessage("{\"value\":false}".getBytes(UTF_8), 1, true, new MqttProperties());
options.setWill(lastWillTopic, will);
```

Elle est **retenue** : quand le courtier la publie au nom du client, elle remplace le `true` périmé. Le composant qui échoue n'est pas celui qui signale l'échec — seul arrangement qui puisse fonctionner. Une réserve : une déconnexion propre ne déclenche **pas** la dernière volonté ; un service bien élevé publie donc `online=false` lui-même en partant, la volonté couvrant les morts brutales.

9.7 Ce qui a survécu au transport, et le couplage restant

La moitié inférieure du tableau système de fichiers→MQTT est l'essentiel : `I`, `S`, `E`, les services égoïstes, la topologie à l'exécution et « transformateur si nécessaire » sont **inchangés**. L'architecture a survécu au remplacement du transport, et c'est la meilleure preuve disponible qu'elle n'a jamais dépendu du transport.

Un couplage faible ne signifie pas que rien ne sait rien. Ce qui reste, nécessairement, est l'accord sur : la syntaxe des messages, leur sémantique, les contrats de sujets, l'Intent, la propriété, le sens du State et celui des Events. Le but n'est pas d'éliminer le couplage mais d'**éliminer la connaissance inutile**. Concrètement, chaque service ici doit comprendre exactement un contrat d'Event étranger — ROT13 comprend `E/text` de ConsoleIn, ConsoleOut comprend `E/crypted` de ROT13 — plus le schéma d'Intent qu'il accepte sur son propre `I`. Cette liste est le minimum irréductible.

Par ailleurs, le courtier est une dépendance d'infrastructure partagée : tout le monde a besoin de son adresse et rien ne marche sans lui. Réel, mais d'une autre nature que la connaissance d'un composant par un autre.

La liste de l'étape 78 est l'outil à garder. Douze questions, dont ces six portent l'argument de ce système :

question	réponse ici
une capacité contient-elle l'identité d'un producteur en dur ?	non — elle arrive par Intent
quels contrats étrangers doit-elle comprendre ?	un contrat d'Event chacun, plus son propre schéma d'Intent
où la topologie est-elle définie ?	dans le configurateur, hors du chemin des données
la topologie peut-elle changer sans recompiler ?	oui — démontré sur un système en marche
le transport pourrait-il être remplacé ?	oui — démontré : système de fichiers puis MQTT
chaque médiateur fait-il quelque chose qui exige une médiation ?	oui, trivialement : `TEXT` → `TEXT` n'en exige aucune, et il n'y en a aucun

La prudence du matériel s'applique : les réponses n'ont pas à être toutes « non ». Elles doivent être **intentionnelles**.

9.8 Le parcours complet

Le programme n'a jamais changé : une ligne entre, la même ressort décalée de 13 lettres. Ce qui a changé, onze fois, c'est la manière dont les parties sont reliées et qui détient la connaissance de ce câblage.

étape	comment les parties sont reliées	où vit la topologie	dans le chemin des données, outre les capacités
1 procédural	appels directs	dans le code de la capacité	—
2 services égoïstes	pas reliées du tout	nulle part — c'est le problème	—

étape	comment les parties sont reliées	où vit la topologie	dans le chemin des données, outre les capacités
3 médiation statique, une JVM	boîtes, copiées par des médiateurs	les médiateurs, démarrés dans `main()`	deux médiateurs
4 médiation statique, système de fichiers	répertoires, copiés par des processus médiateurs	les lignes de commande de l'opérateur	deux processus médiateurs
5 médiation statique, Syncthing	répertoires répliqués	les lignes de commande de l'opérateur	deux processus médiateurs, plus le délai de réplication
6 médiateurs nommés et configurables	Intent `ROUTE` d'un orchestrateur	l'orchestrateur, hors du chemin	les médiateurs, encore
7 dynamique, une JVM	les services écoutent, guidés par un Intent `TOPIC`	le configurateur	rien
8 dynamique, système de fichiers	idem, sur des répertoires	le configurateur	rien
9 dynamique, Syncthing	idem, sur des répertoires répliqués	le configurateur	rien — seulement des sauts de transport
10 contrats	compatibles : direct ; incompatibles : un transformateur	le configurateur	un transformateur, seulement là où les contrats diffèrent
11 MQTT	des sujets sur un courtier	le configurateur	rien — le courtier est du transport

Lisez la troisième colonne de haut en bas : la topologie s'éloigne, une étape après l'autre, jusqu'à ce qu'aucune capacité n'en détienne plus rien. Lisez la dernière : le prix de ce déplacement tombe à zéro à l'étape 7 et y reste pendant que le transport change trois fois en dessous. Même architecture, meilleur transport.

9.9 Les phrases à retenir

- **« Event-driven » décrit comment circulent l'information et le contrôle.**
- **Le couplage faible décrit ce que les participants doivent savoir les uns des autres.**
- **Publiez ce qui est arrivé.**
- **Déclarez ce que vous voulez.**
- **Publiez votre propre état.**
- **Gardez les capacités égoïstes.**
- **Gardez la topologie hors des capacités.**
- **Ne payez pas un médiateur qui ne fait pas réellement de médiation.**

Et chaque fois qu'une architecture devient confuse, revenez à la première question de ce livre : **qui connaît qui ?**

Glossaire

Chaque terme ci-dessous est défini là où il sert pour la première fois, et lié depuis chaque mention ultérieure. La flèche d'une entrée ramène à cet endroit.