Contrôle de processus

Romain De Groote

6 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 **4**
Comment lire les figures 4

1 Étape 0 — De quoi il s'agit **5**
1.1 Les trois questions 5

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

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

4 Partie III — Le rendre réellement distribué **17**
4.1 Syncthing réplique ; il ne distribue pas 17
4.2 Tester une couche à la fois 17
4.3 Ce que coûte une frappe 17
4.4 Casser un médiateur 18
4.5 Le médiateur devient un service ordinaire 18
4.6 Ce que la partie III a déplacé 19

5 Partie IV — Rembobinage **20**
5.1 Le raccourci évident, et son piège 20
5.2 La capacité n'est pas la topologie 20
5.3 Ce qui change dans le code 20
5.4 `ROUTE` et `TOPIC` sont de l'Intent 21

Annexes **22**

Glossaire **22**

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.

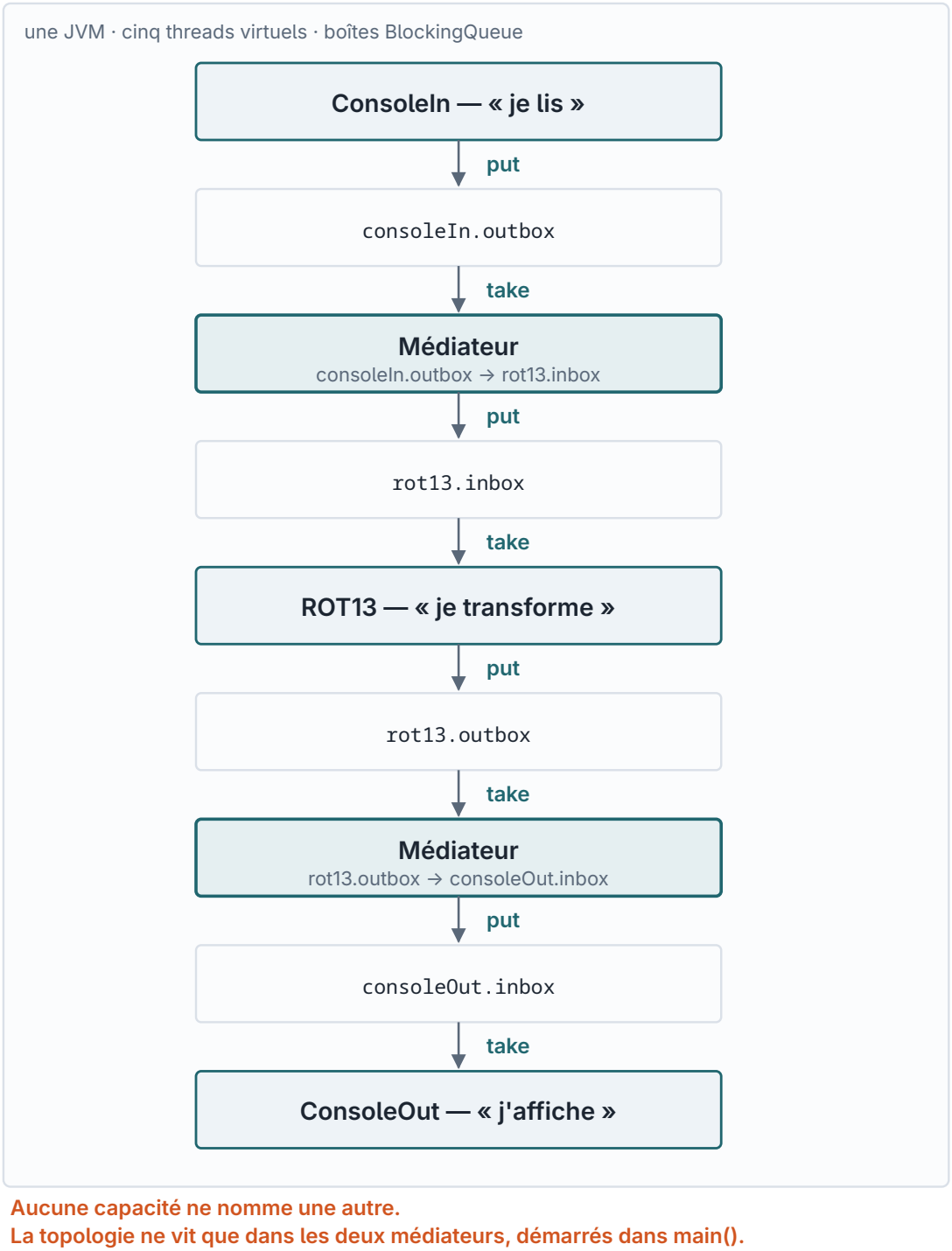


Figure 2.2: La version médiatisée dans une seule JVM. Dans la version web, **Lancer** fait parcourir à un message les huit sauts.

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.

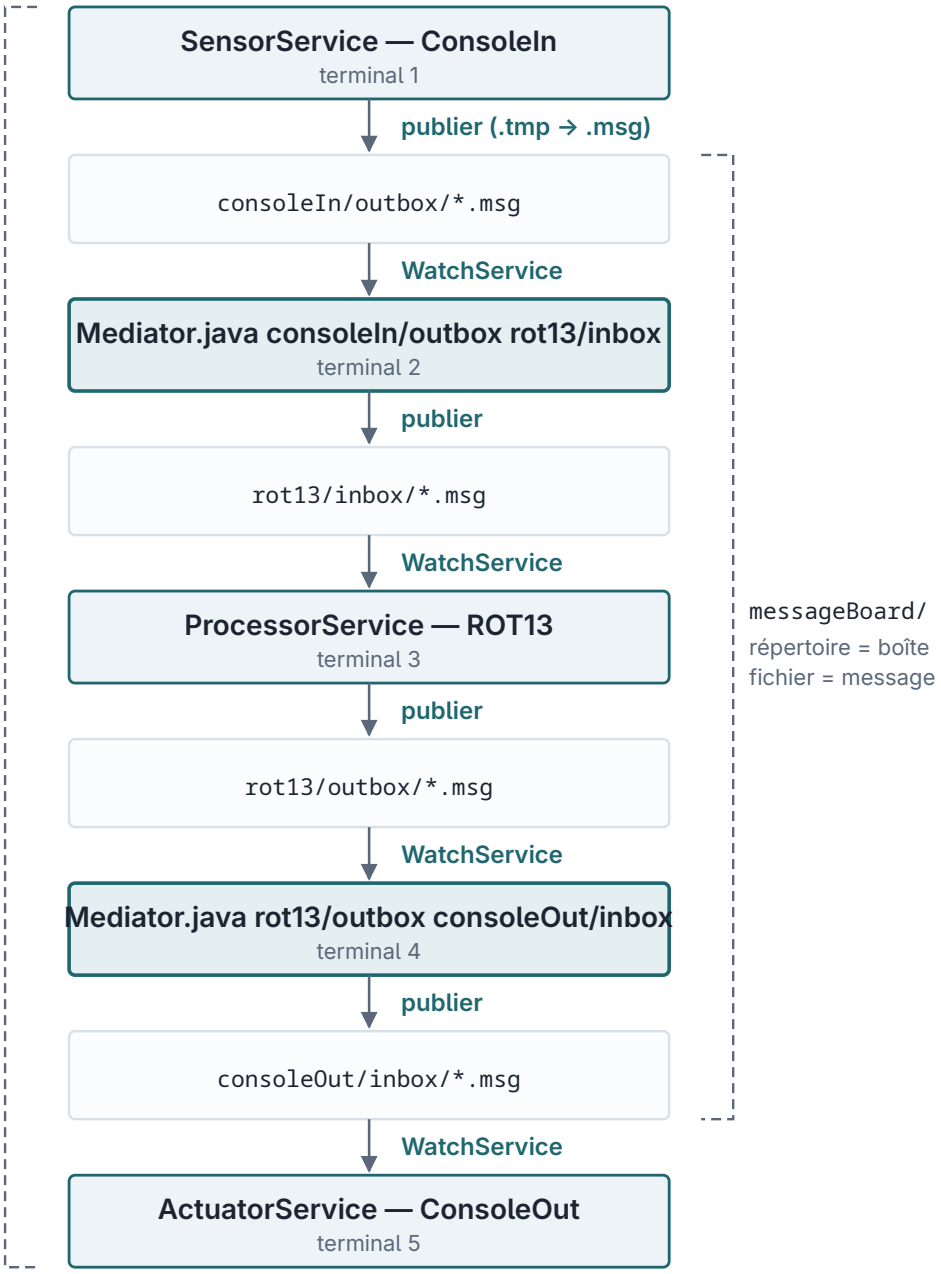


Figure 3.1: Les cinq processus et les quatre répertoires de boîtes. Dans la version web, **Lancer** suit une ligne de texte du clavier à la console.

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 qu'au cours d'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)
```

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.